Hardware Equivalence & Property Verification

Jie-Hong Roland Jiang

National Taiwan University

Outline

- Introduction
  - Motivations
  - Systems to be verified
    - Hardware vs. software
  - Verification methodologies
    - Formal vs. informal verification
  - Verification formalisms
    - Temporal logics vs. model checking
  - Properties to be verified
    - Safety vs. liveness
- Computation basics
  - Data structures and Boolean reasoning engines
- Equivalence checking
  - Combinational and sequential EC
    - Structure-based verification
    - Function-based verification
- Safety property checking
  - Bounded and unbounded model checking
    - *k*-step induction
    - Interpolation

Introduction

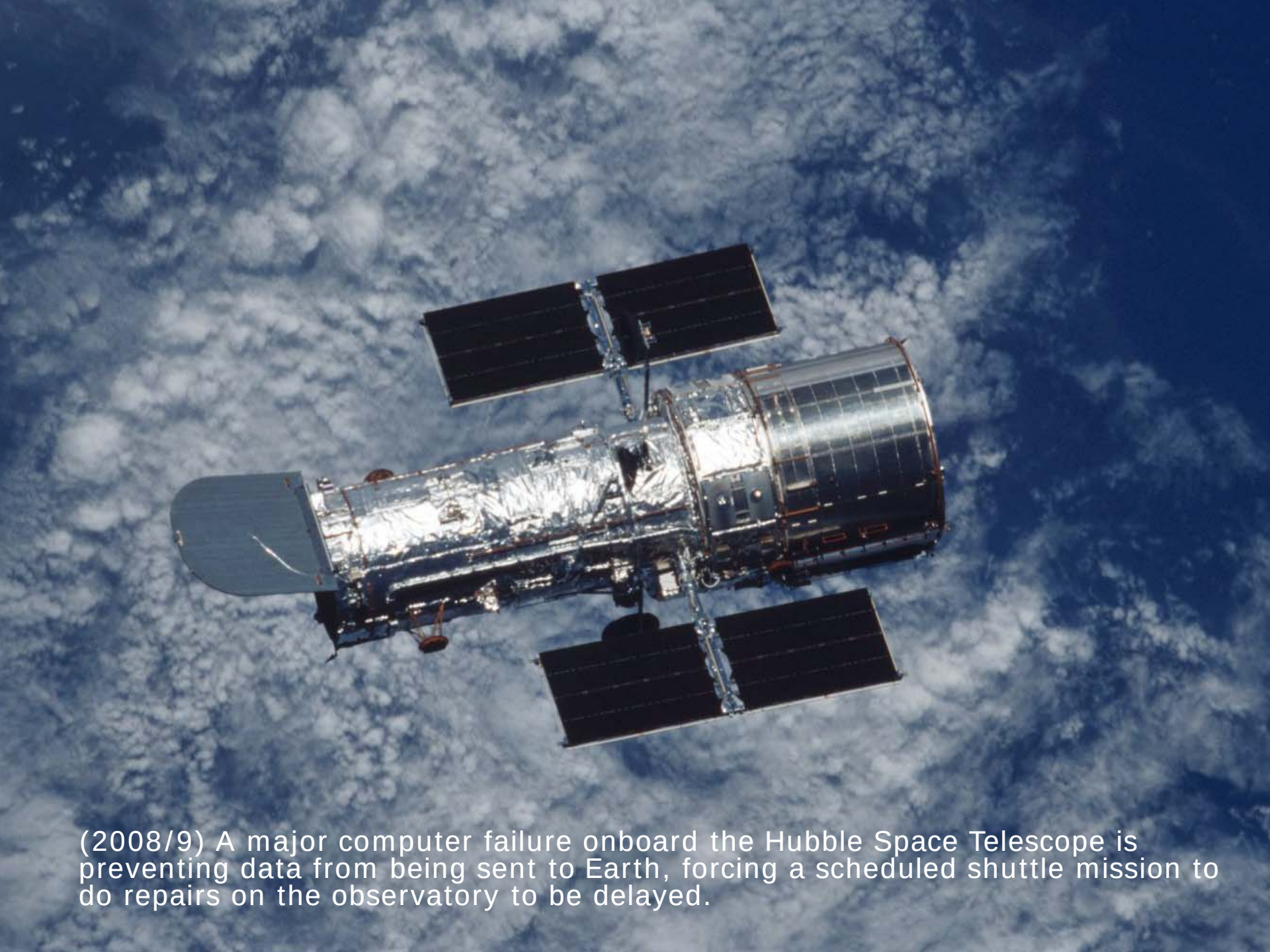
Motivations

- Costs of system failures
- Computational hardness

(1995/1) Intel announces a pre-tax charge of 475 million dollars against earnings, ostensibly the total cost associated with replacement of the flawed processors.

(1996/6) The European Ariane5 rocket explodes 40 s into its maiden flight due to a software bug.

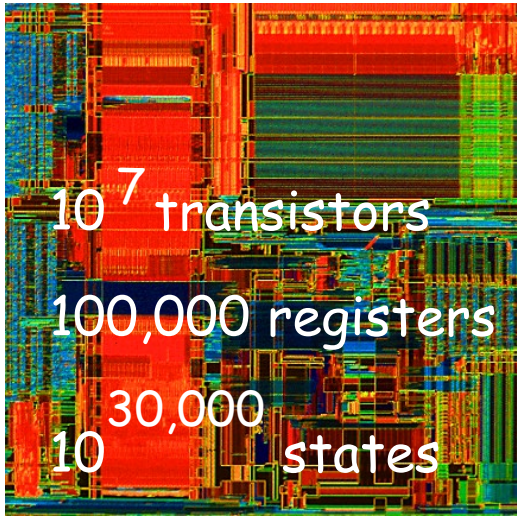
(2003/8) A programming error has been identified as the cause of the Northeast power blackout, which affected an estimated 10 million people in Canada and 45 million people in the U.S.

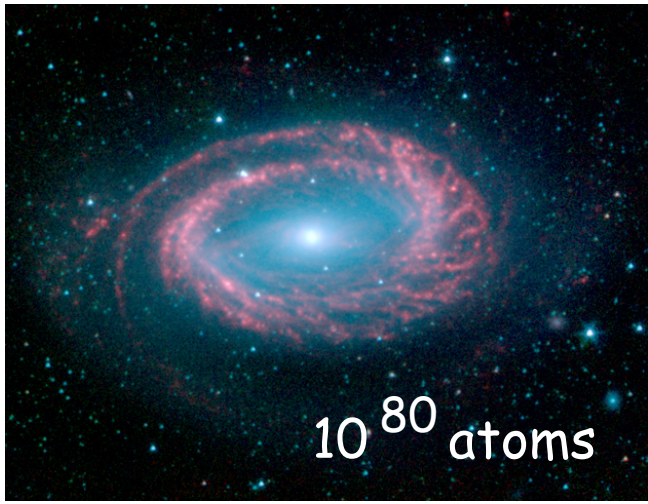


(2008/9) A major computer failure onboard the Hubble Space Telescope is preventing data from being sent to Earth, forcing a scheduled shuttle mission to do repairs on the observatory to be delayed.

Hardness

- Verification may take 70% of the entire design cycle of a system
- State explosion problem
  - #states is exponential in #registers (state-holding elements)





Systems to Be Verified

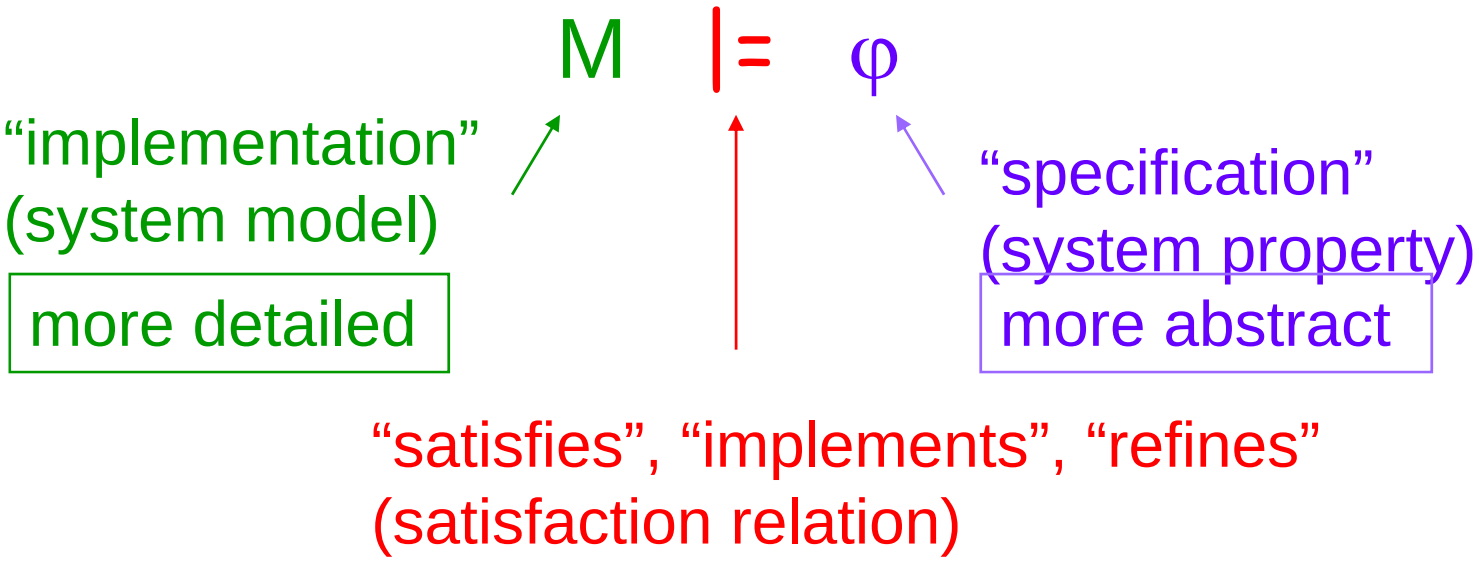
- **Hardware** vs. software
  - Finite state vs. infinite state
    - Hardware systems can be modeled as finite-state transition systems
    - Software systems are often modeled as infinite-state transition systems

Verification Methodologies

- Informal vs. **formal**
  - Informal
    - Incomplete
      - E.g., by software simulation or hardware emulation
    - Useful in finding bugs, but not in showing the absence of bugs
  - Formal
    - Complete
      - E.g., theorem proving, property checking, equivalence checking
    - Useful in both debugging and proving correctness

Verification Formalisms

- Temporal logics vs. **model checking**
  - Temporal logics are useful specifying temporal properties
    - E.g., may (branching time) vs. must (linear time)
    - Not the only way of specifying properties
  - Model checking is an automatic procedure checking whether a model of a system satisfies a given specification



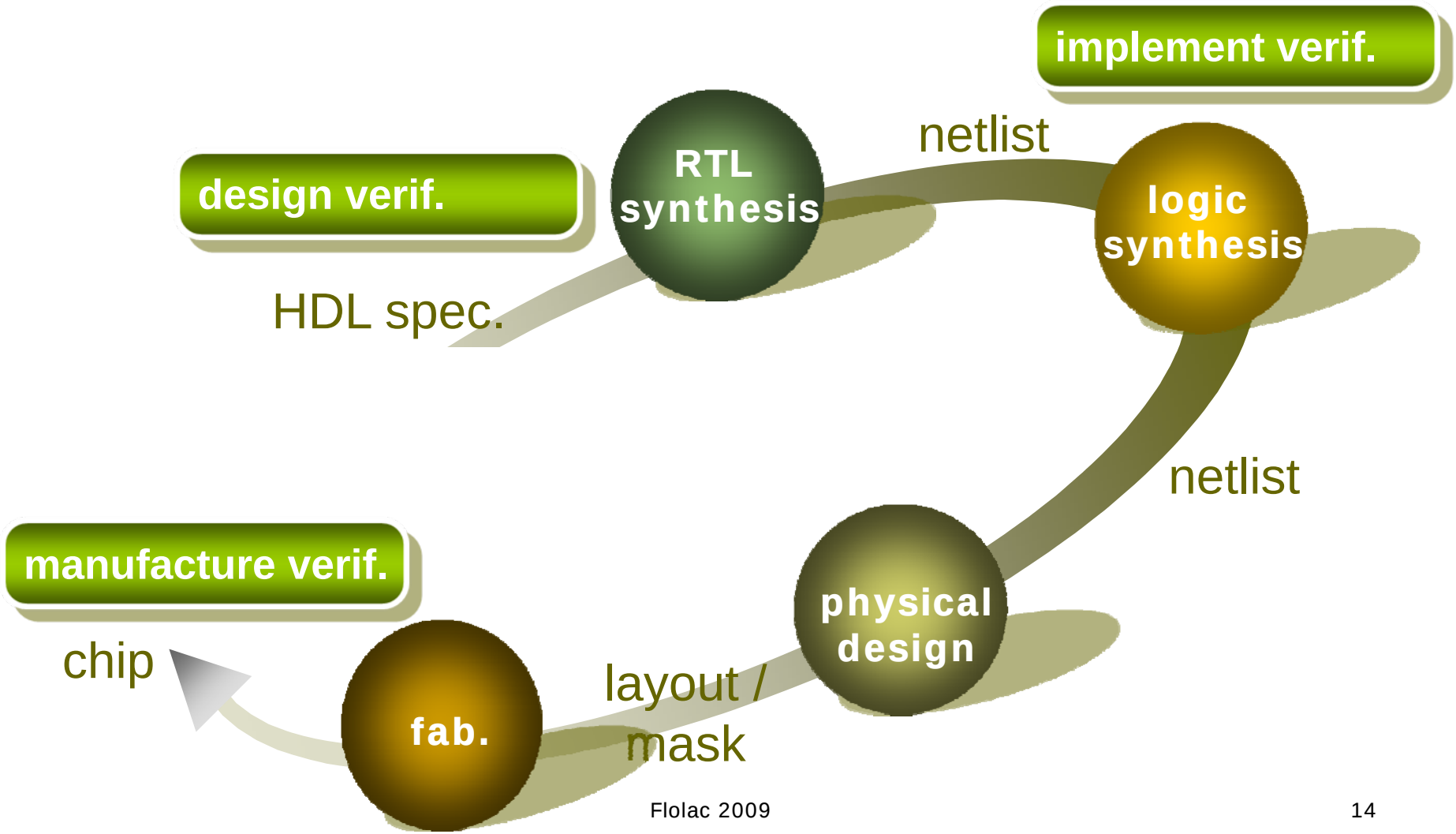
Properties to Be Verified

- **Safety** vs. liveness
  - Safety property
    - Something bad will never happen

      couterexample of finite length
  - Liveness property
    - Something good will happen eventually or infinitely often

      counterexample of infinite length
  - 90% of the verification problems are checking safety properties
  - Liveness property checking can be converted to safety property checking for finite state systems

IC Design Flow and Verification



Hardware Verification

- **Design verification**
  - Does a design specification satisfy some properties?
  - Property checking / assertion-based verification
- **Implementation verification**
  - Does an implementation conform to the original specification?
  - Equivalence checking / (design rule checking)
- Manufacture verification
  - Does a manufactured design have no defects?
  - Testing

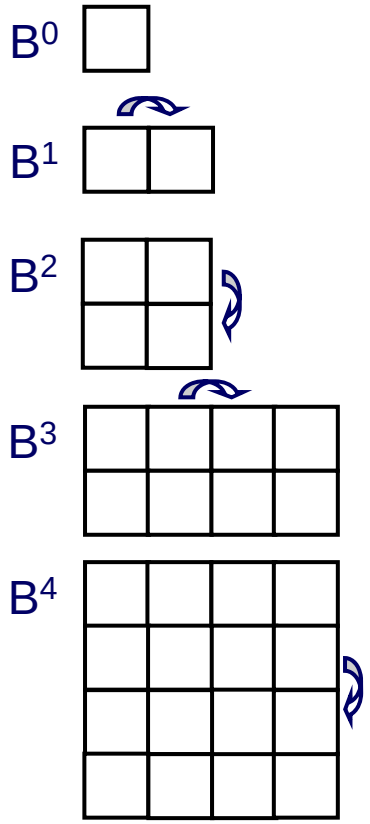
Computation Basics

Boolean Space

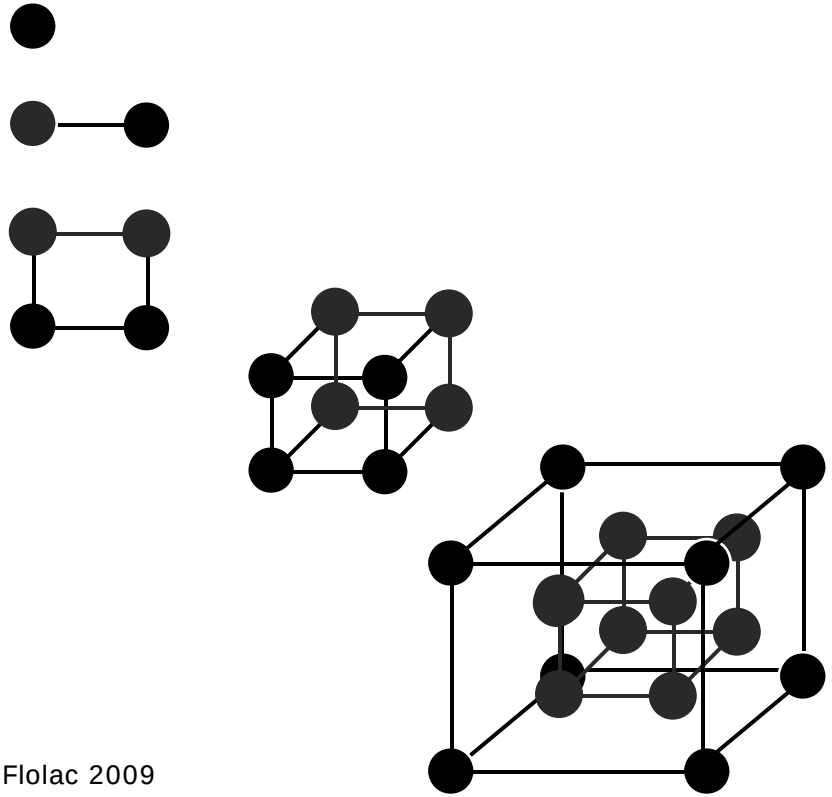
$B = \{0,1\}$

$B^2 = \{0,1\} \times \{0,1\} = \{00, 01, 10, 11\}$

Karnaugh Maps:



Boolean Lattices:

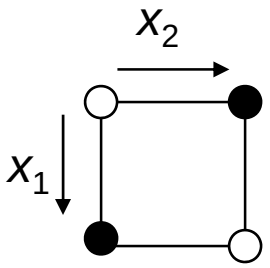


Boolean Functions

- A Boolean function f: $B^n \rightarrow B$ over variables $x_1, x_2, \ldots, x_n$ maps each Boolean valuation (truth assignment) in B^n to either 0 or 1
  - E.g. $f(x_1, x_2)$

$x_1 x_2$	f
0 0	1
0 1	0
1 0	1
1 1	0

x_1 | x_2	0	1
1	1	0



- The output value of f partitions B^n into two sets

  onset ($f = 1$):
  - E.g. {00, 10} (i.e., with characteristic function $F^1 = \neg x_2$)

  offset ($f = 0$):
  - E.g. {01, 11} (i.e., with characteristic function $F^0 = x_2$)

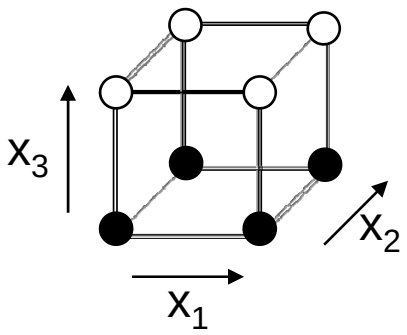
  - A **literal** is a Boolean variable x or its negation $\neg x$ in a Boolean formula

Boolean Functions

- The **onset** of f, denoted as F^1, is $F^1 = \{v \in B^n \mid f(v) = 1\}$
  - If $F^1 = B^n$, f is a *tautology*
- The **offset** of f, denoted as F^0, is $F^0 = \{v \in B^n \mid f(v) = 0\}$
  - If $F^0 = B^n$, f is *unsatisfiable*. Otherwise, f is *satisfiable*.
- Two Boolean functions f and g are *equivalent* if $\forall v \in B^n.\ f(v) \equiv g(v)$

Boolean Functions

- There are 2^n vertices in Boolean space B^n
- There are 2^{2^n} distinct n-variable Boolean functions
  - Each $F^1 \subseteq B^n$ corresponds to a distinct Boolean function



$x_1x_2x_3$	
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	$\Rightarrow$ 1
1 0 1	0
1 1 0	1
1 1 1	0

Boolean Operations

Given two Boolean functions:

f: $B^n \rightarrow B$

g: $B^n \rightarrow B$

- $h = f \wedge g$ from **conjunction** is defined as
  $H^1 = F^1 \cap G^1$; $H^0 = B^n \setminus H^1$
- $h = f \vee g$ from **disjunction** is defined as
  $H^1 = F^1 \cup G^1$; $H^0 = B^n \setminus H^1$
- $h = \neg f$ from **complement** is defined as
  $H^1 = F^0$; $H^0 = F^1$

Cofactor & Quantification

Given a Boolean function:

$f : B^n \rightarrow B$, with input variables $(x_1,\ldots,x_i,\ldots,x_n)$

- **Positive cofactor**, $h = f_{xi}$, is defined as
  $h = f(x_1,\ldots,1,\ldots,x_n)$
- **Negative cofactor**, $h = f_{\neg xi}$, is defined as
  $h = f(x_1,\ldots,0,\ldots,x_n)$
- **Existential quantification** over variable x_i, $h = \exists x_i.\ f$, is defined as
  $h = f(x_1,\ldots,0,\ldots,x_n) \vee f(x_1,\ldots,1,\ldots,x_n)$
- **Universal quantification** over variable x_i, $h = \forall x_i.\ f$, is defined as
  $h = f(x_1,\ldots,0,\ldots,x_n) \wedge f(x_1,\ldots,1,\ldots,x_n)$
- **Boolean difference** over variable x_i, $h = \partial f/\partial x_i$, is defined as
  $h = f(x_1,\ldots,0,\ldots,x_n) \oplus f(x_1,\ldots,1,\ldots,x_n)$

Data Structures

- Basic data structures for Boolean function representation
  - Truth tables
  - Binary Decision Diagrams (BDDs)
  - AND-INV graphs (AIGs)
  - Conjunctive Normal Forms (CNFs)
  - ...
- Why bother having different data structures?

Data Structures

Data-structure revolution in verification

- State graph (late 70s-80s)
  - Problem size ~10^4 states
- BDD (late 80s-90s)
  - Problem size ~10^{20} states
  - Critical resource: memory
- SAT (late 90s-)
  - GRASP, SATO, chaff, berkmin
  - Problem size ~10^{100} (?) states
  - Critical resource: CPU time

Data Structures – BDDs

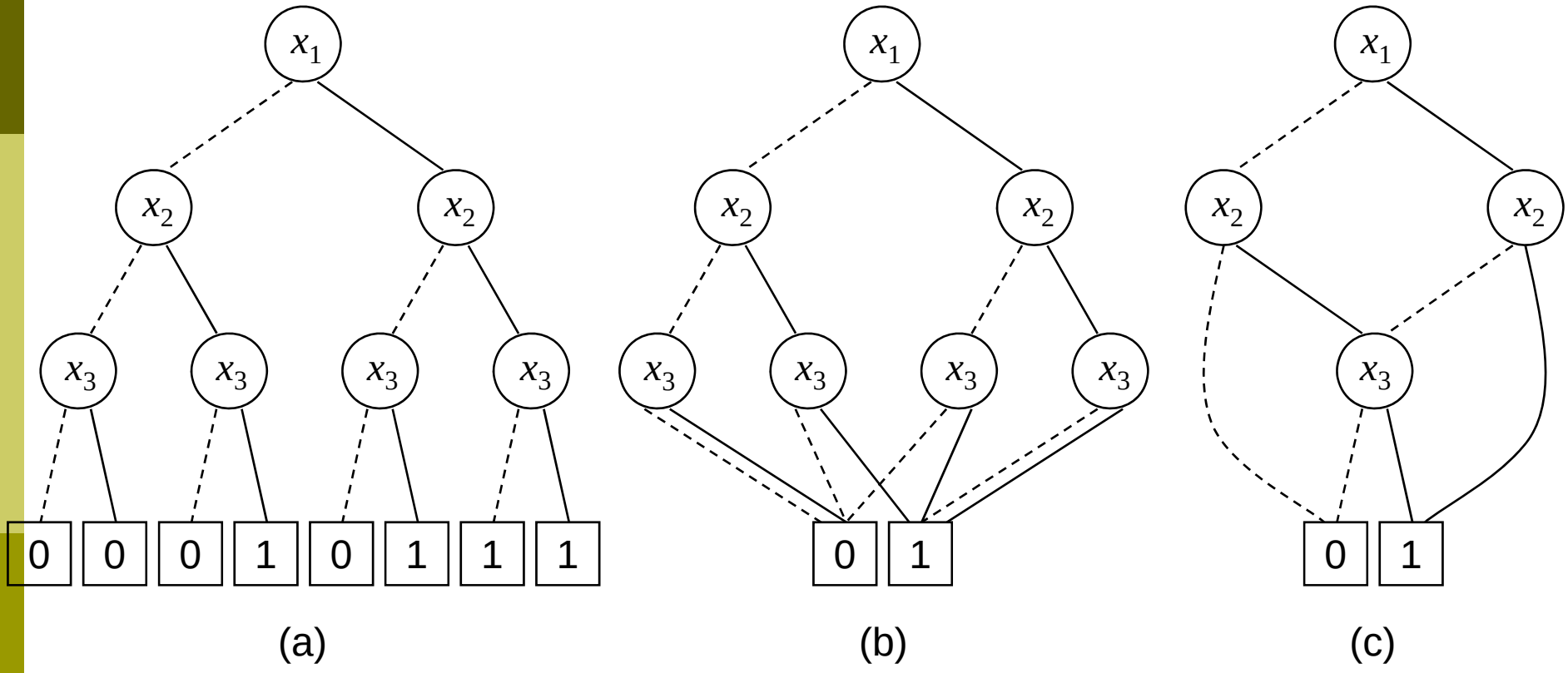
- BDDs are graph representations of Boolean functions
  - A non-terminal node is a decision node (multiplexer) controlled by some variable *v*
    - It represents some Boolean function *f*
    - Its two children represent two functions f_v and $f_{v'}$
    - They together represent a Shannon cofactor tree
      $f = v\, f_v + v'\, f_{v'}$ (Shannon expansion)
  - A terminal node is either constant “0” or “1”

Data Structures – BDDs

- Reduced Ordered BDDs (ROBDDs)
  - Ordered:
    - Variables follow the same order along all paths

      $x_{i_1} < x_{i_2} < x_{i_3} < \ldots < x_{i_n}$
  - Reduced:
    - Any node with two identical children is removed
    - Two nodes with isomorphic BDD's are merged
  - These two rules make any node of an ROBDD represent a distinct function and make ROBDDs *canonical* representation of Boolean functions

Data Structures – BDDs

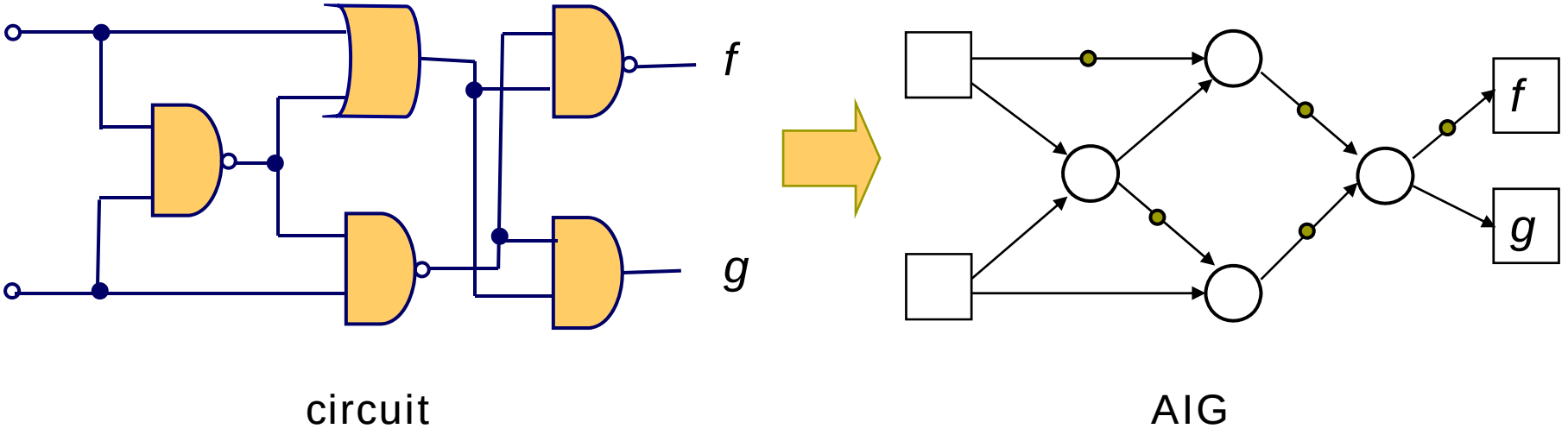


Ordered BDDs of $f = x_1x_2 + x_1x_2'x_3 + x_1'x_2x_3$

Data Structures – AIGs

- AND-INV graphs (AIGs)
  - vertices:
    - 2-input AND gates
  - edges:
    - interconnects with (optional) dots representing INVs
  - {AND, INV} is a *functionally complete* set of Boolean operators
  - Structurally isomorphic nodes can be merged

Data Structures – AIGs



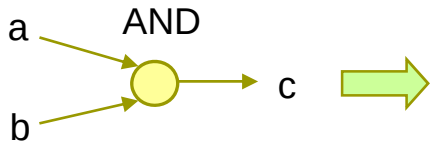
Data Structures – SAT

- Conjunctive Normal Form (CNF)
  - Product of sums

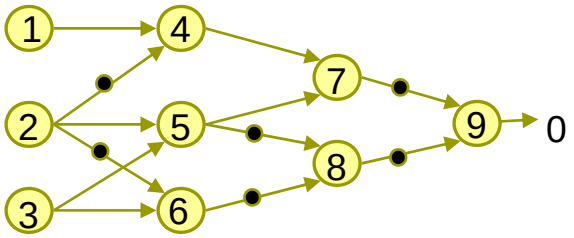
    e.g., $\varphi = (a+b'+c)(a'+b+c)(a+b'+c')(a+b+c)$
  - CNF is useful for satisfiability (SAT) checking

Data Structures – SAT

- Circuit-to-CNF conversion



$$(\neg a + \neg b + c)(a + \neg c)(b + \neg c)$$



Is output always 0 ?

Justify to “1”

$$(\neg 1 + 2 + 4)(1 + \neg 4)(\neg 2 + \neg 4)$$
$$(\neg 2 + \neg 3 + 5)(2 + \neg 5)(3 + \neg 5)$$
$$(2 + \neg 3 + 6)(\neg 2 + \neg 6)(3 + \neg 6)$$
$$(\neg 4 + \neg 5 + 7)(4 + \neg 7)(5 + \neg 7)$$
$$(5 + 6 + 8)(\neg 5 + \neg 8)(\neg 6 + \neg 8)$$
$$(7 + 8 + 9)(\neg 7 + \neg 9)(\neg 8 + \neg 9)$$
$$(9)$$

Conversion can be done in time linear to the circuit size!

Boolean Reasoning

- A Boolean function can be represented in different forms
  - E.g., BDD, AIG, CNF, …
- Boolean reasoning studies the intrinsic characteristics of a Boolean function
  - We may be interested in characteristics such as *satisfiability*, *validity*, *decomposability*, etc., of a function
- There are different Boolean reasoning engines based on different data structures
  - E.g. BDD packages, AIG packages, SAT solvers

Boolean Function Manipulation

- **Characteristic functions**
  - Functional representations of "sets"
    - Predicates indicating whether an element is in a set
  - Operations over sets (union, intersection, complement) become Boolean operations (OR, AND, INV) over characteristic functions

E.g.,

Let $X=\{000,001,110,111\}$ and $Y=\{001,101,110\}$
(assume B^3 is our universal set)

Their characteristic functions are
$f_X = x_1'x_2'+x_1x_2$, $f_Y = x_1'x_2+x_1x_2x_3'$

The set $X \cup Y$ has characteristic function $f_X \vee f_Y$
The set $X \cap Y$ has characteristic function $f_X \wedge f_Y$

Equivalence Checking

Digital Circuits

- Combinational circuits
  - Implement Boolean functions
  - Have no state-holding elements (registers)
- Sequential circuits
  - Implement finite state machines
  - Have state-holding elements
- Combinational circuits can be considered as single-state sequential circuits

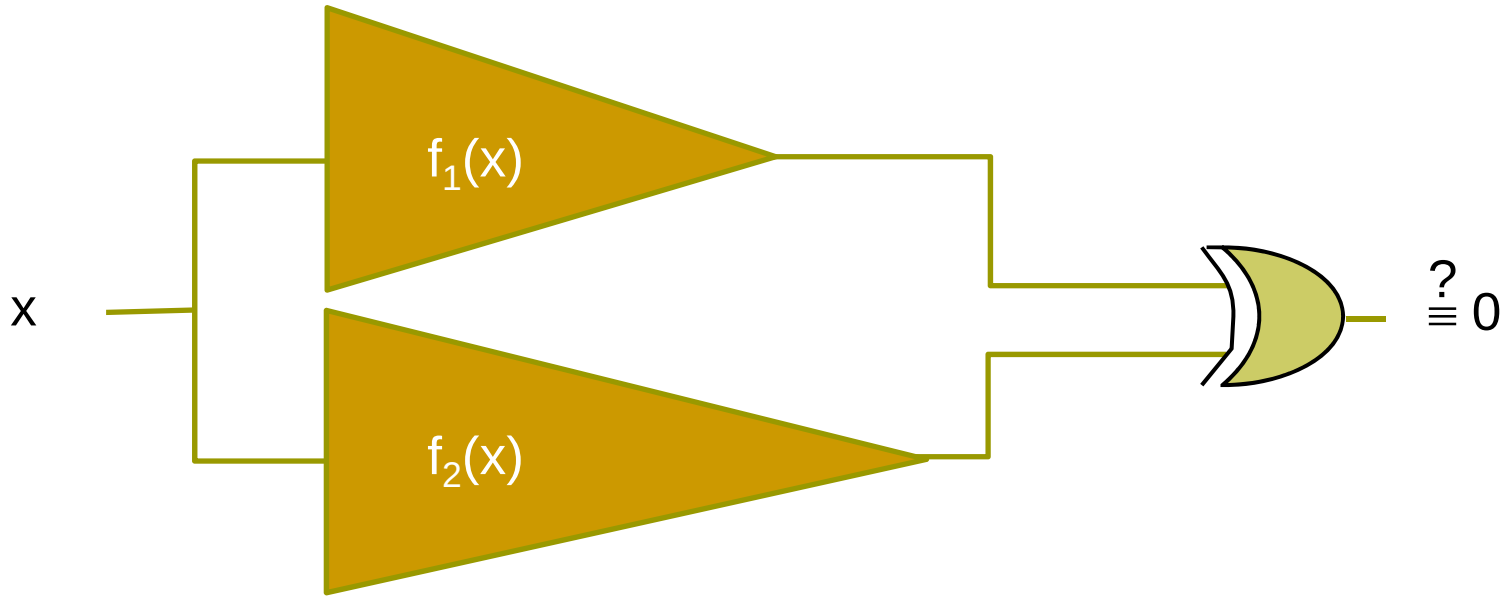
Equivalence Checking

- Combinational EC
  - Check if two combinational circuits are equivalent, i.e., if they have the same input-output behavior under all *input assignments*
- Sequential EC
  - Check if two sequential circuits are equivalent, i.e., if they have the same input-output behavior under all *input sequences*

Hardness

- Hardness of verification
  - Combinational EC is coNP-complete
  - Sequential EC and safety property checking are PSPACE-complete

Combinational EC



To check if the two circuits implementing f_1 and f_2 are equivalent, we build their **miter**

They are equivalent iff the miter circuit is equivalent to a constant-0 function (can be formulated as SAT solving!)

Combinational EC

- BDD-based computation
  1. Construct the ROBDDs of f_1 and f_2
     - Variable orderings of f_1 and f_2 should be the same
  2. Let $g = f_1 \oplus f_2$ equals constant 0 iff the two circuits are equivalent

Combinational EC

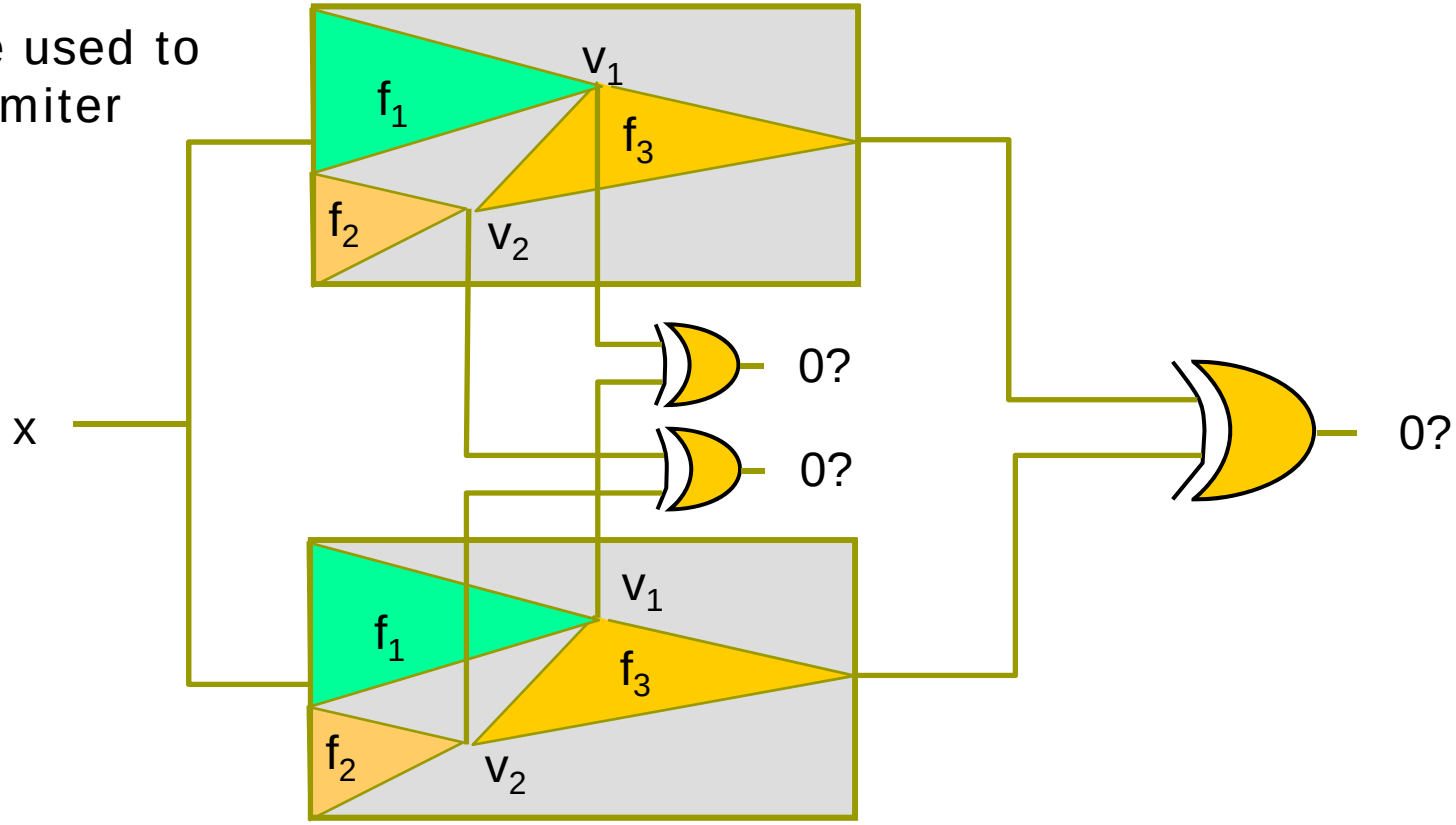
- SAT-based computation
  1. Convert the miter structure into a CNF
  2. Perform SAT solving to verify if the output variable cannot be valuated to true under all input assignments (i.e., unsatisfiable)

Combinational EC

- Pure BDD and plain SAT solving cannot handle large CEC problems
- To be scalable, contemporary methods highly exploit structural similarities between two circuits to be compared
  - Identify and merge *cutpoints* (identical internal signals)

Combinational EC

Cutpoints are used to partition the miter

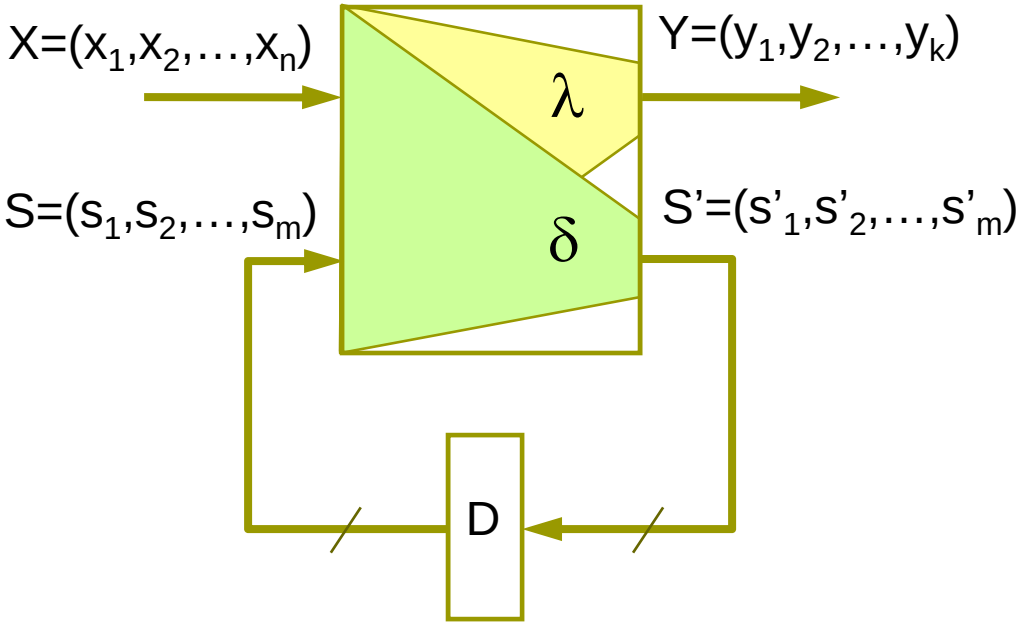


Successively merge equivalent signals from inputs to outputs to simplify the EC problem

Combinational EC

- Solved in most industrial circuits (w/ multi-million gates)
  - Computational efforts scale almost linearly with the design size
  - Existence of structural similarities
    - Logic transformations preserve similarities to some extent
  - Hybrid engine of BDD, SAT, AIG, simulation, etc.
    - Cutpoint identification
- Unsolved for arithmetic circuits
  - Absence of structural similarities
    - Commutativity ruins internal similarities
  - Word- vs. bit-level verification

Finite State Machines



M([[X]],[[Y]],[[S]],I,δ,λ):

[[X]]: Input alphabet

[[Y]]: Output alphabet

[[S]]: State set

I : Initial state(s)

δ : $[[X]] \times [[S]] \rightarrow [[S]]$
(next-state function or transition function)

λ : $[[X]] \times [[S]] \rightarrow [[Y]]$
(output function)

State Transition Systems

- Transition function vs. transition relation
  - Transition function:
    Transition must be *deterministic* (there is a unique next state for any current state and input)
  - Transition relation:
    Transition may be *nondeterministic* (there can be a several next states for any current state and input)

- Conversion from transition functions ($\delta_1,\ldots,\delta_n$) to a transition relation T

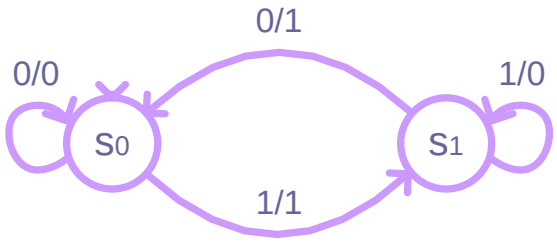
$$T(\vec{x},\vec{s},\vec{s}') = \bigwedge_{i=1}^{n}(s'_i \equiv \delta_i(\vec{x},\vec{s}))$$

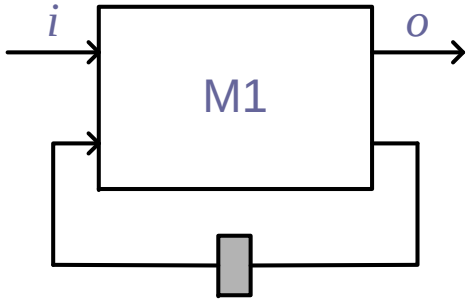
When we are interested in reachability only, we may further quantify the inputs

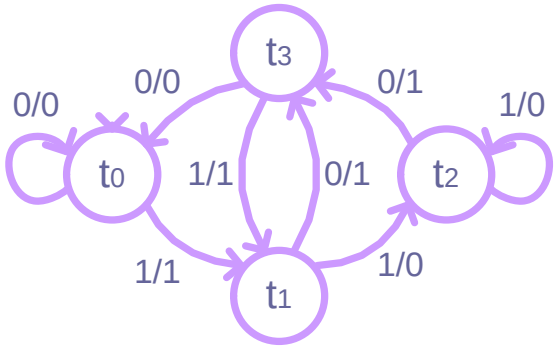
$$T_{\exists}(\vec{s},\vec{s}') = \exists\vec{x}[\bigwedge_{i=1}^{n}(s'_i \equiv \delta_i(\vec{x},\vec{s}))]$$

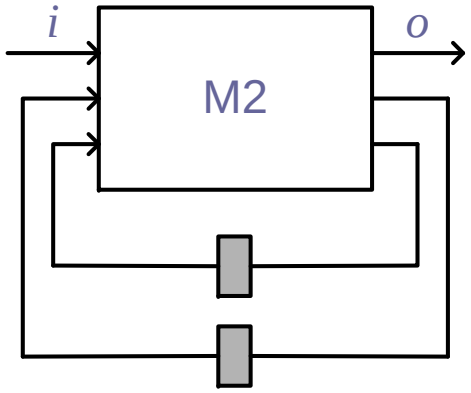
Sequential EC

- Combinational checking for sequential equivalence is sound, but not complete (may yield false-negative)
  - Equivalent FSMs may have different state transitions and encodings

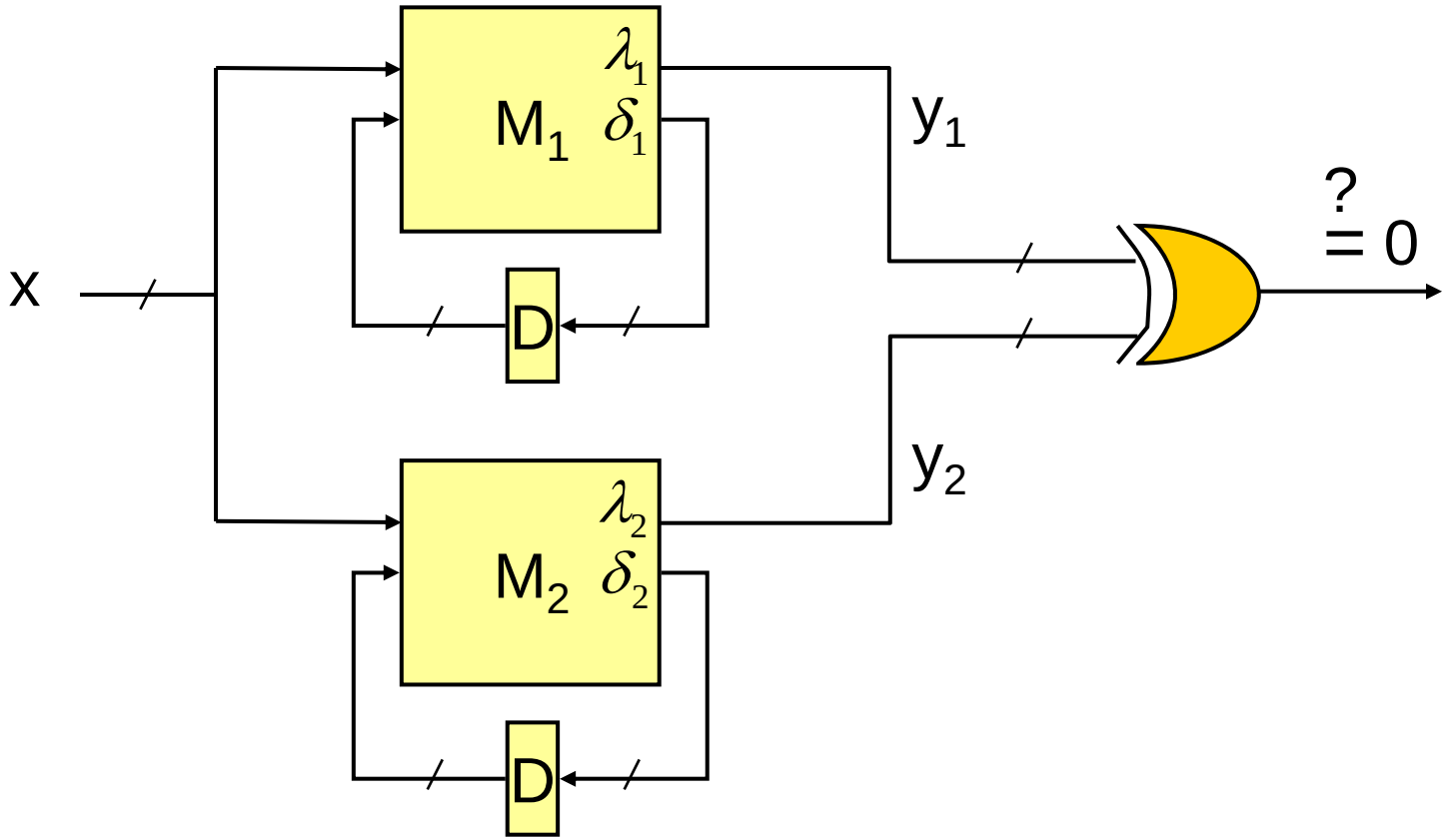








Sequential EC



Two FSMs M1 and M2 are equivalent if and only if the output of their product machine always produces constant 0

Product Machine

- The product FSM $M_{1\times2}$ of FSMs

  $M_1 = ([[X]], [[Y_1]], [[S_1]], I_1, \delta_1, \lambda_1)$ and

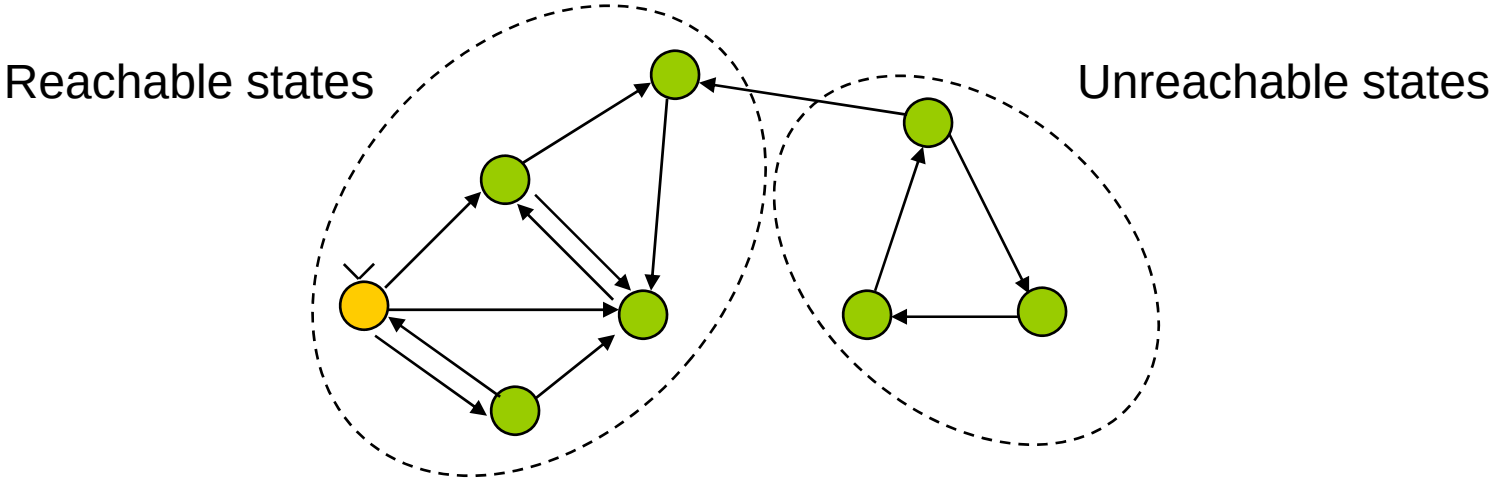
  $M_2 = ([[X]], [[Y_2]], [[S_2]], I_2, \delta_2, \lambda_2)$ has
  - State space $[[S_1]] \times [[S_2]]$
  - Initial state set $I_1 \times I_2$
  - Input alphabet $[[X]]$
  - Output alphabet $\{0,1\}$
  - Transition function $\delta_{1\times2} = (\delta_1, \delta_2)$
  - Output function $\lambda_{1\times2} = (\lambda_1 \oplus \lambda_2)$

Sequential EC

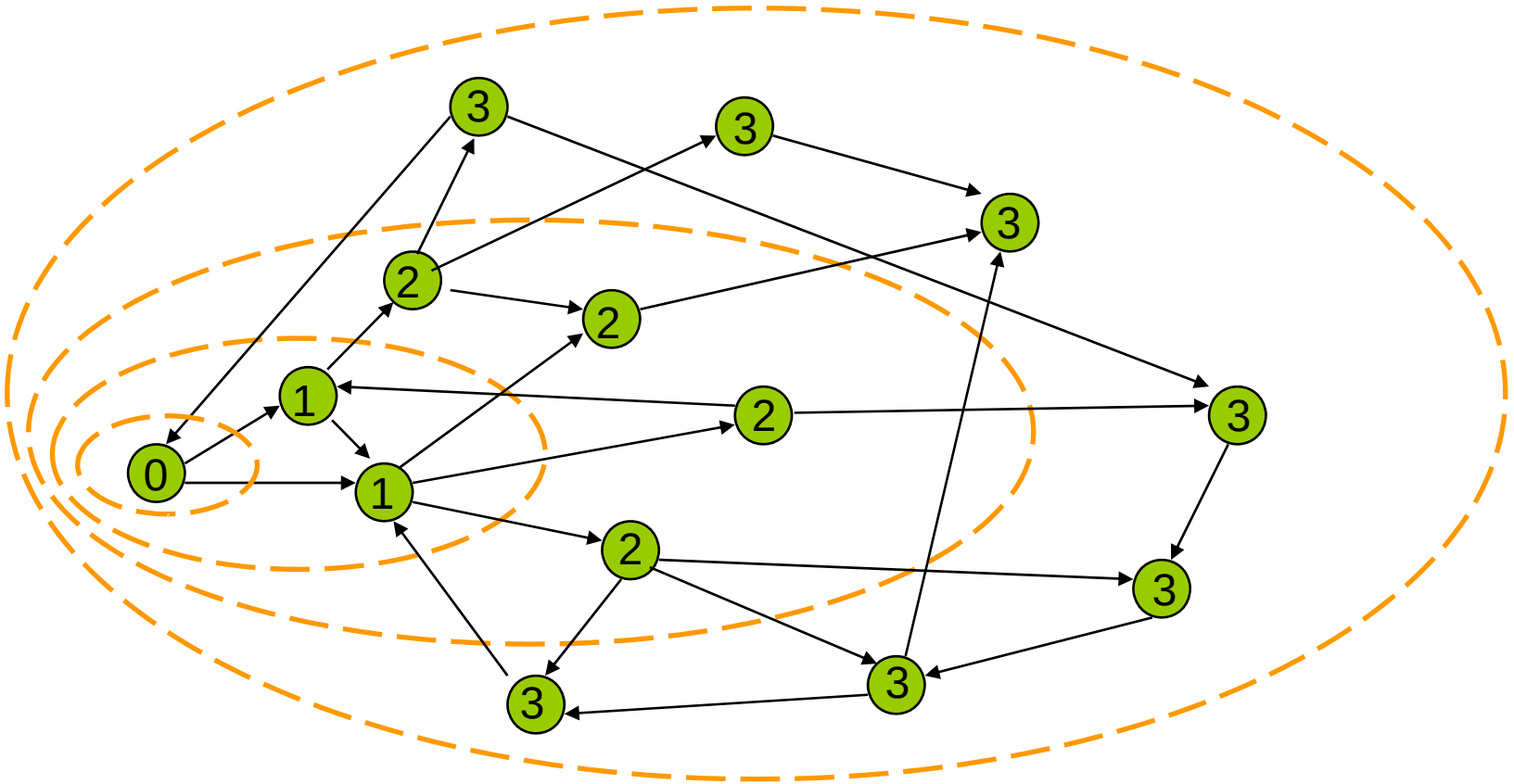
- When the reachable states of the product machine is known, SEC reduces to CEC!
  - Let R be the characteristic function of the reachable state set and , T_1 and T_2 be the transition relations of M_1 and M_2
  - M_1 and M_2 are equivalent iff $(\lambda_{1\times 2} \wedge R)$ is unsatisfiable
    - There is no state that is both bad and reachable
- So the main computation of SEC is **reachability analysis**

Reachability Analysis

- Given an FSM, which states are reachable from the initial state?



Reachability "Onion Rings"



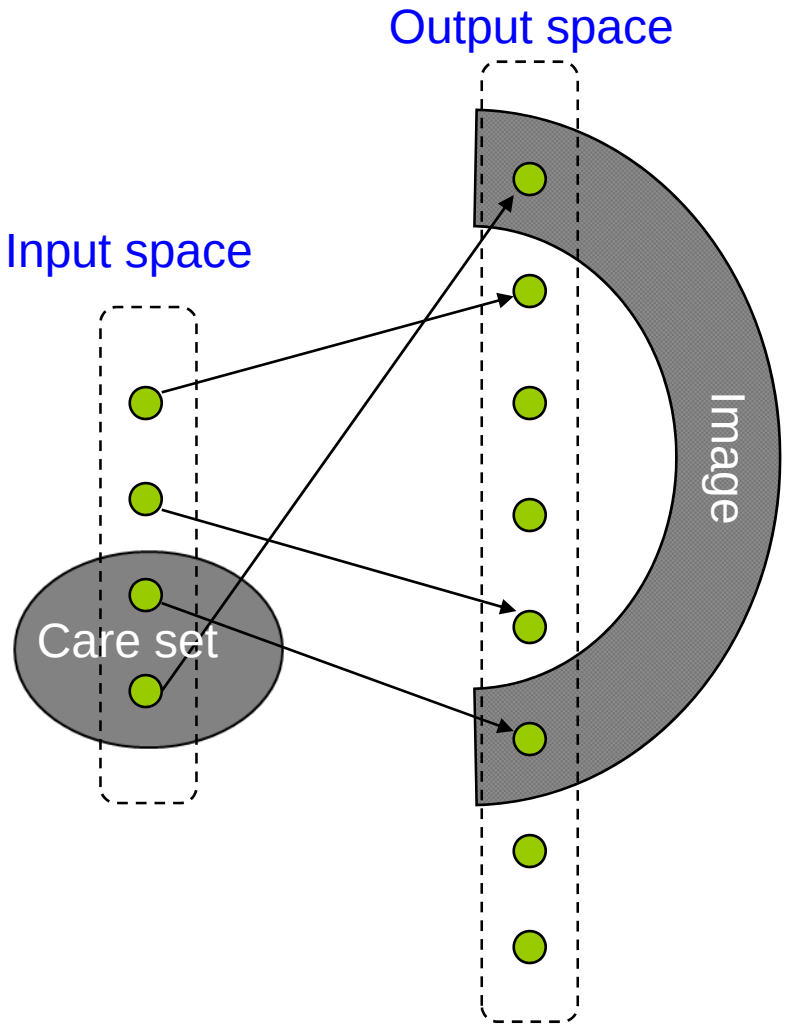
Symbolic Reachability Analysis

- Reachability analysis can be performed either explicitly (over state transition graphs) or implicitly (over transition functions or relations)
  - Implicit reachability analysis is also called symbolic reachability analysis (often using BDDs and more recently SAT)

- **Image computation** is the core computation in symbolic reachability analysis

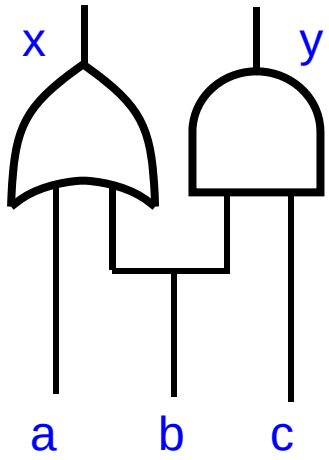
Image Computation

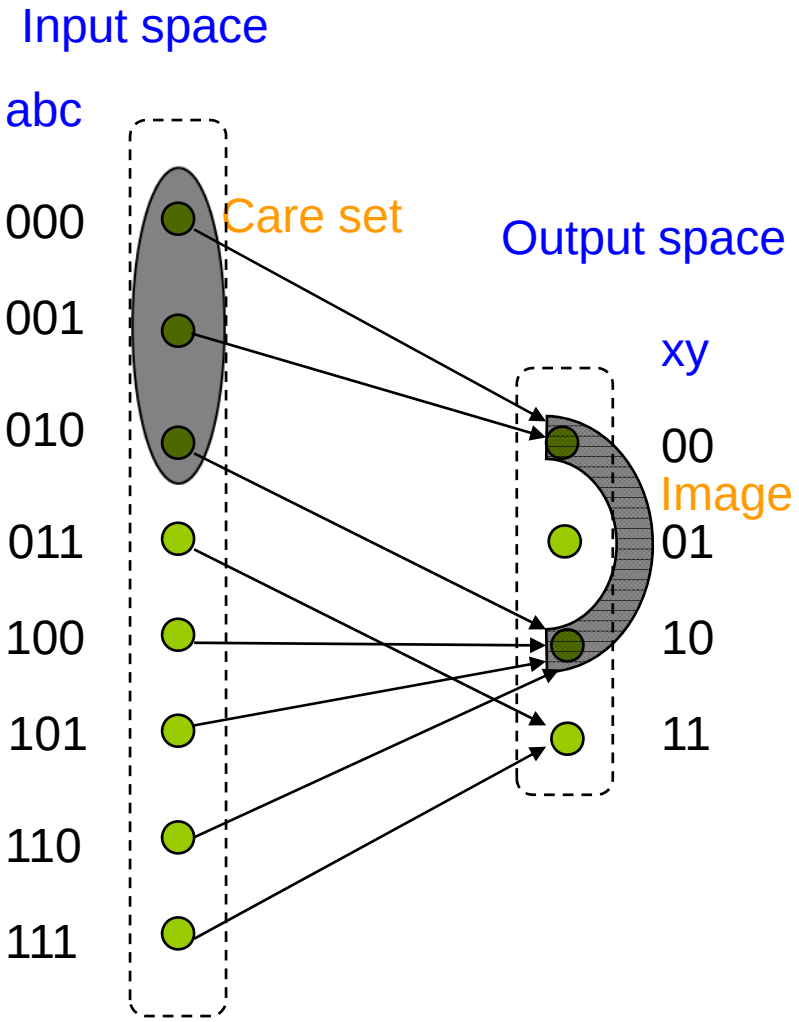
- Given a mapping of one Boolean space (input space) into another Boolean space (output space)
  - For a set of minterms (care set) in the input space
    - The image is the set of related minterms from the output space
  - For a set of minterms in the output space
    - The pre-image is the set of related minterms in the input space



Courtesy of A. Mishchenko

Image Computation





Courtesy of A. Mishchenko

Symbolic Image Computation

- $\mathrm{Img}(C(x),T(x,y)) = \exists x\,[C(x) \wedge T(x,y)]$
  - Image of C under T
- Implicit methods by far outperform explicit ones
  - Successfully compute images with more than 2^{100} minterms in the input/output spaces
- Operations $\wedge$ and $\exists$ are basic Boolean manipulations are implemented using BDDs
  - To avoid large intermediate results (during and after the product computation), operation AND-EXIST is used, which performs product and quantification in one pass over the BDD

Next-State Computation

- What is the set *P* of next-states from *Q*?

$$P(\vec{s}') = Img(Q(\vec{s}), T_{\exists}(\vec{s}, \vec{s}'))$$

$$= \exists \vec{s}.(Q(\vec{s}) \wedge T_{\exists}(\vec{s}, \vec{s}'))$$

Previous-State Computation

- What is the set *P* of previous-states of *Q*?

$$P(\vec{s}) = PreImg(Q(\vec{s}'), T_{\exists}(\vec{s}, \vec{s}'))$$

$$= \exists \vec{s}'.(Q(\vec{s}') \wedge T_{\exists}(\vec{s}, \vec{s}'))$$

Reachability Analysis

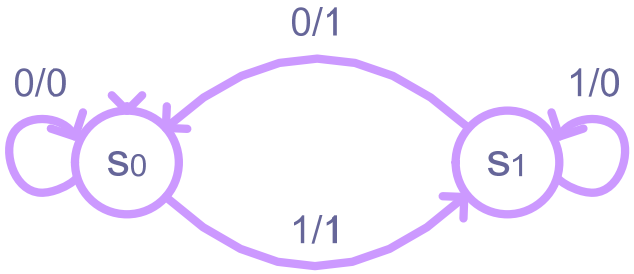
```
ForwardReachability(Transition Relation T, Initial State I )
{
    i := 0
    R^i := I
    repeat
        R_new = Img( R^i, T );
        i := i + 1
        R^i := R^(i-1) ∨ R_new
    until R^i = R^(i-1)
    return R^i
}
```

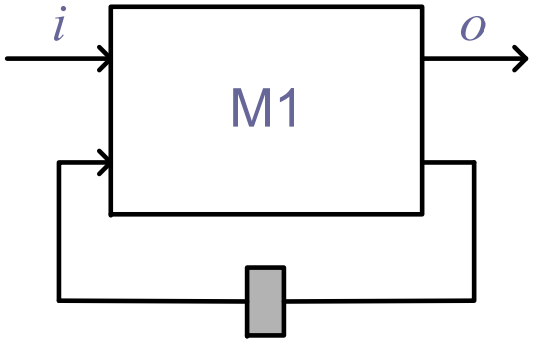
Backward reachability analysis can be done in a similar manner with pre-image computation and starting from final states to see if they can be reached from initial states.

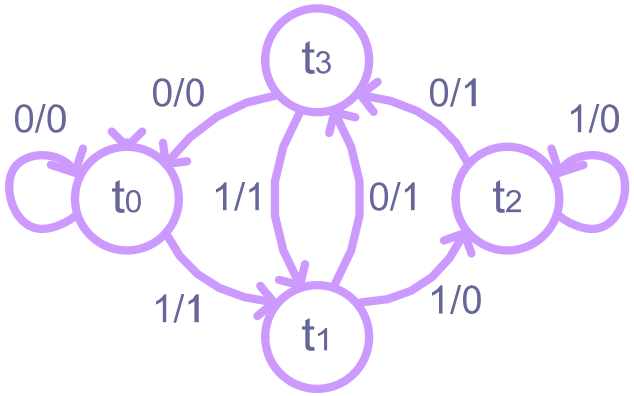
The procedures can be realized using BDD package.

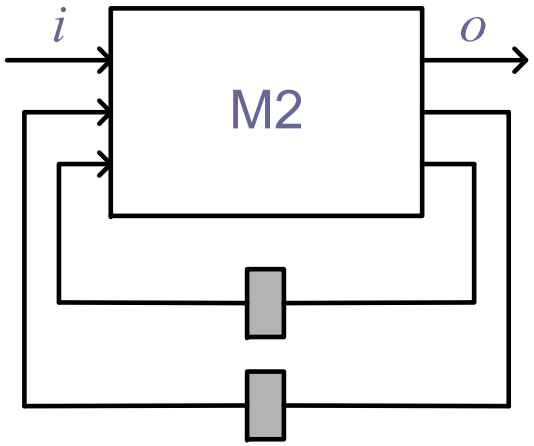
Reachability Analysis

Example





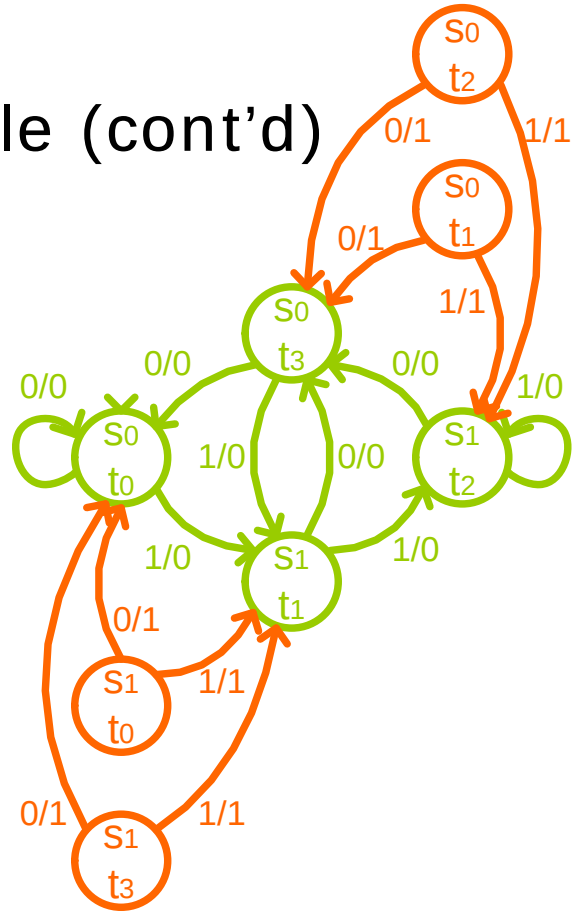


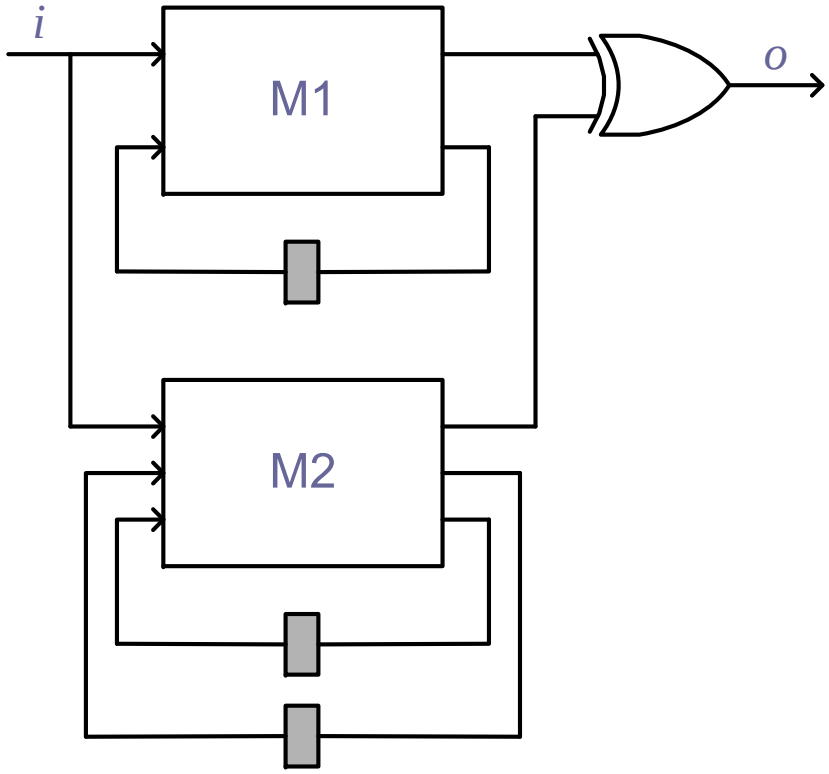


FSMs to be equivalence checked

Reachability Analysis

Example (cont'd)

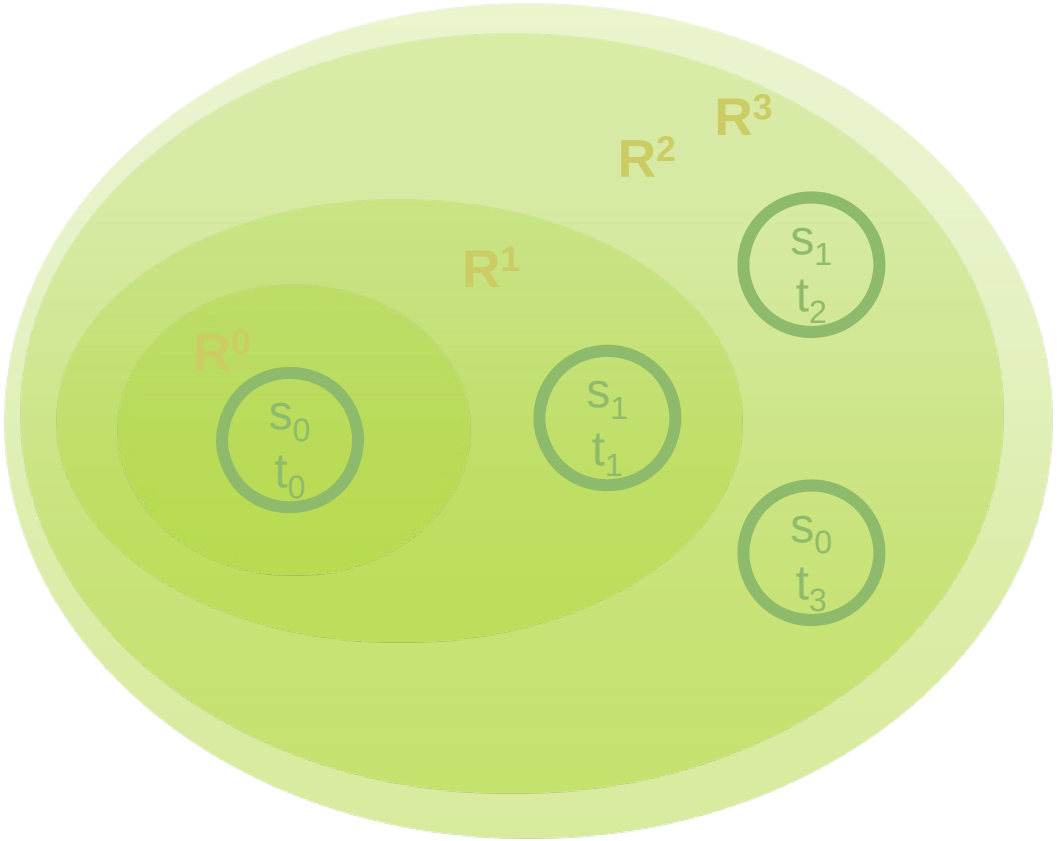


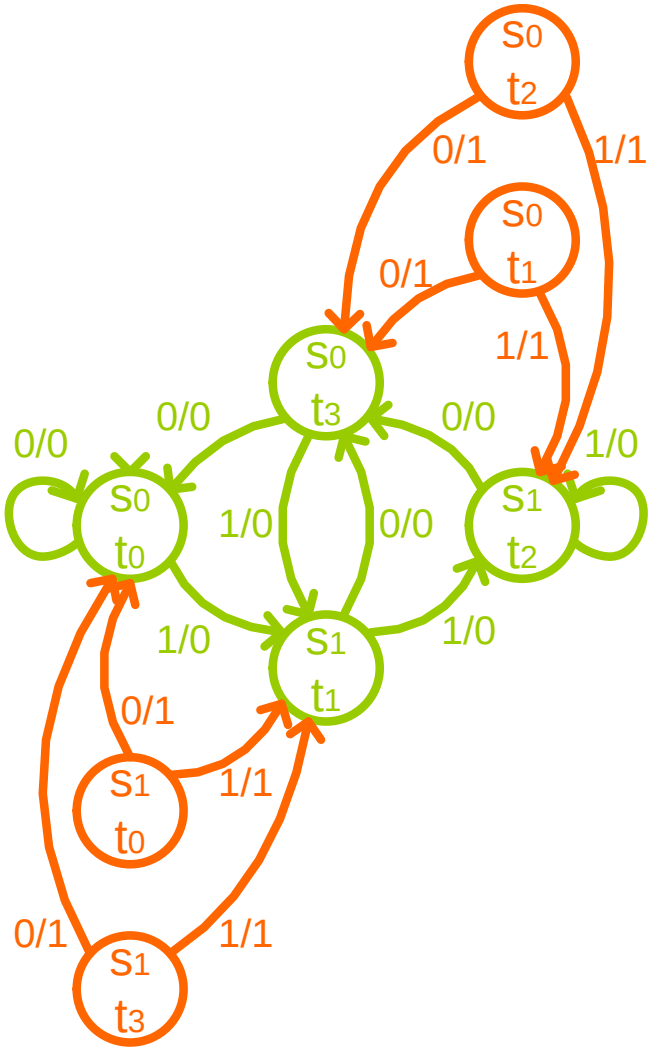


Product FSM and its state transition graph

Forward Reachability Analysis

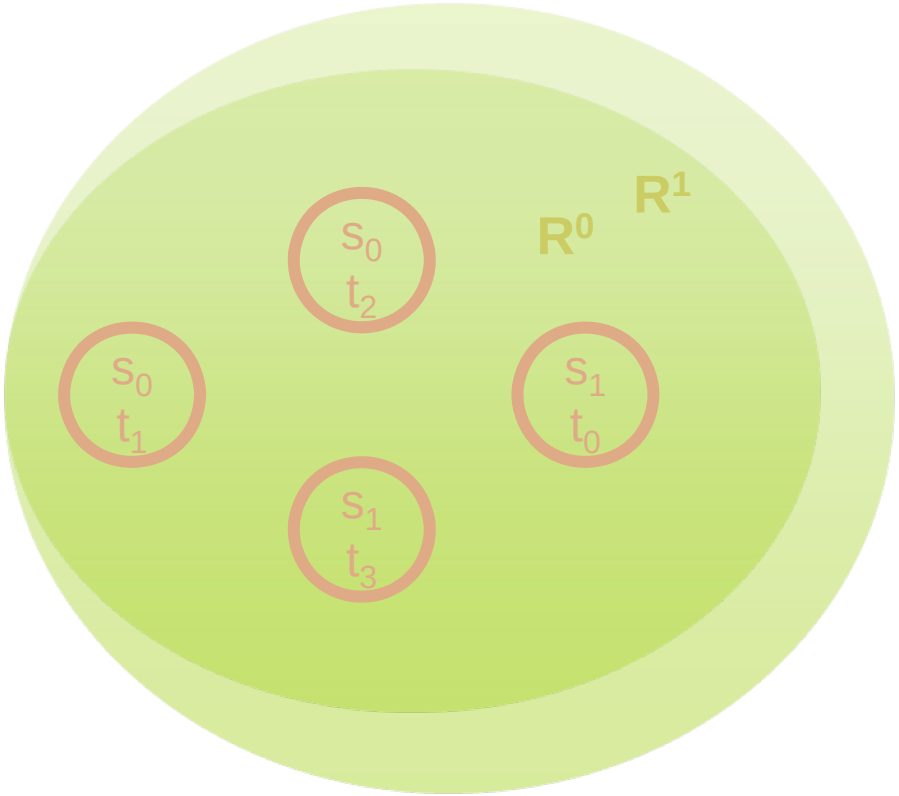
Example (cont'd)

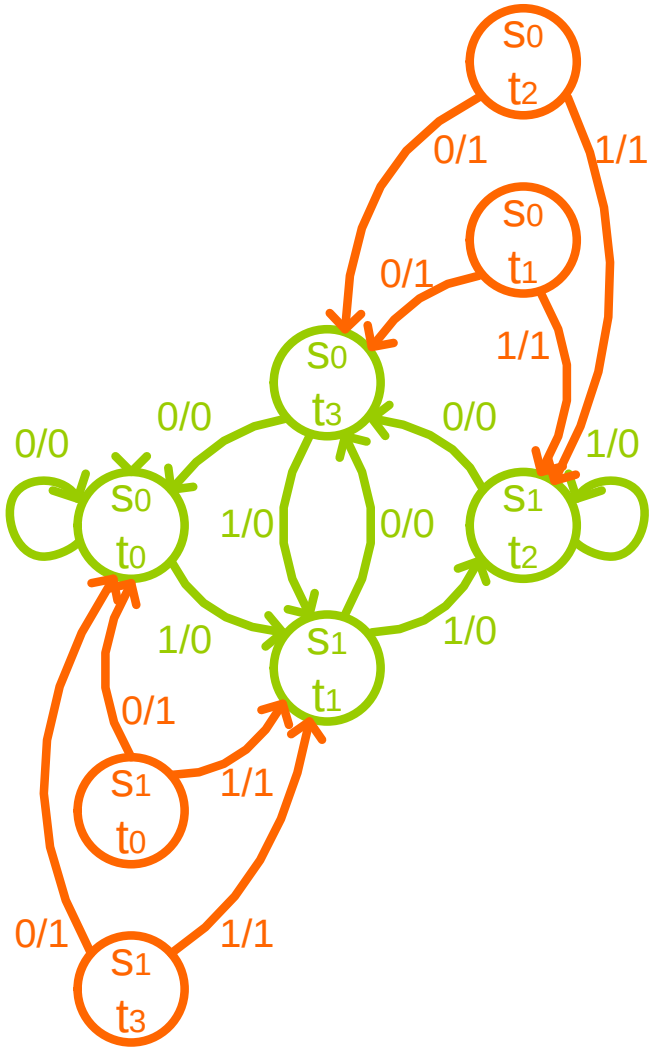




Backward Reachability Analysis

Example (cont'd)



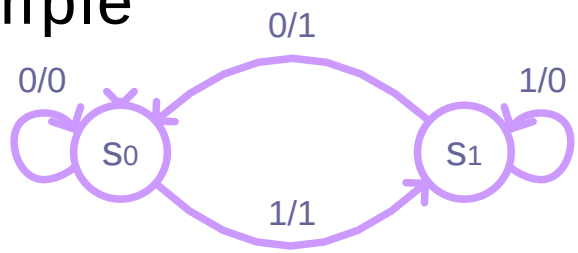


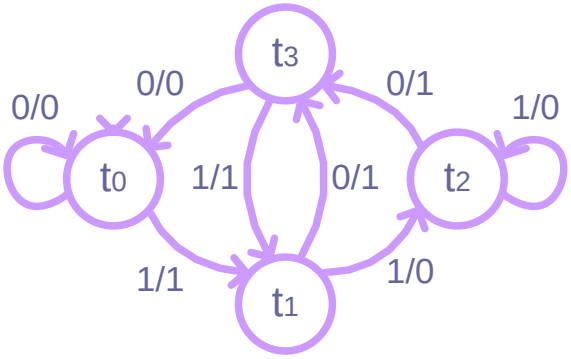
Sequential EC

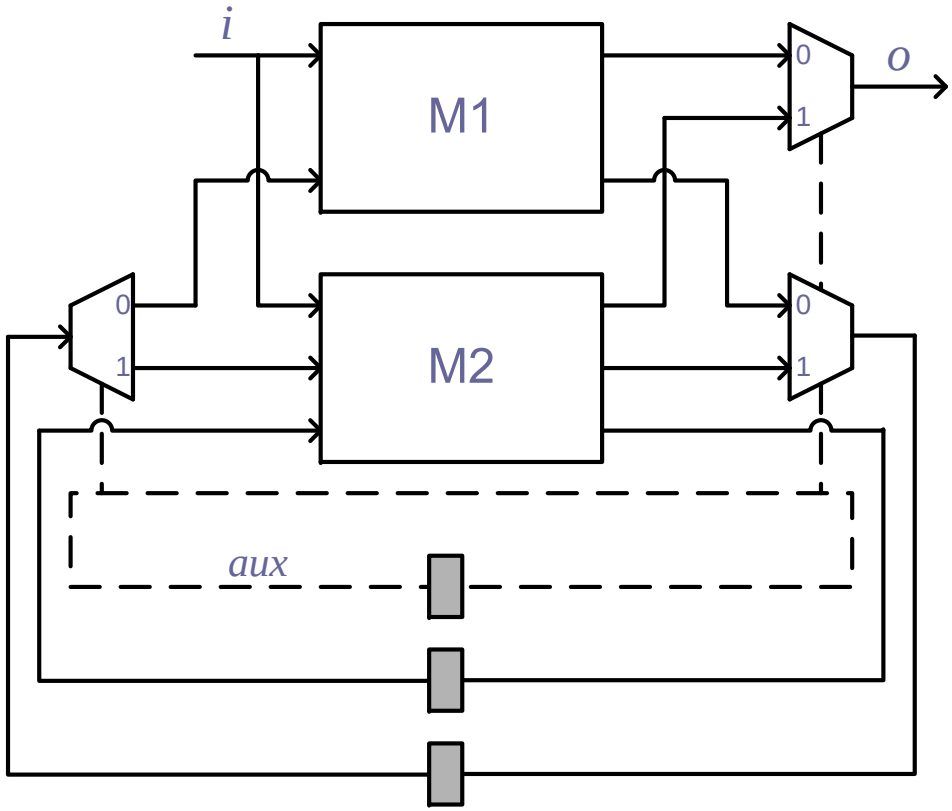
- Reachability analysis (product state space)
  - Explicit traversal on product STG
  - Implicit image computation on product FSM
- State equivalence (disjoint union state space)
  - Explicit equivalence state identification on disjoint union STG
  - Implicit state partitioning on multiplexed FSM

State Partitioning

Example







Multiplexed FSM and the disjoint union STG

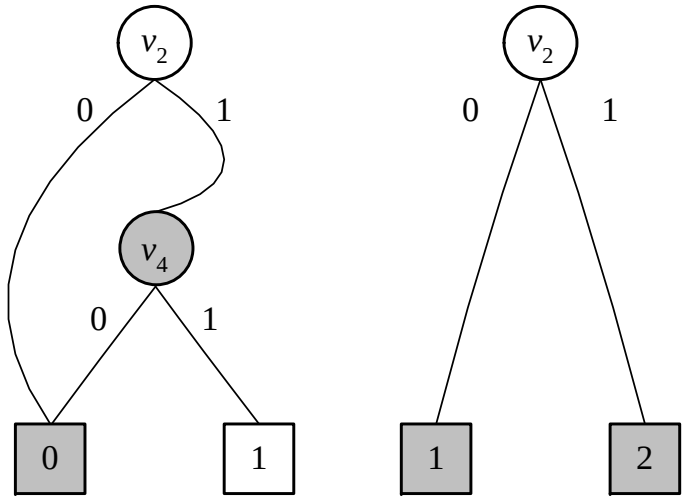
State Partitioning

- BDD-based functional decomposition
  - Bound set variables (top): state variables
  - Free set variables (bottom): others
  - Cutset: free-set nodes with incoming edges from bound-set nodes
- Paths leading to a node in the cutset form an equivalence class of states (for an iteration)
- Iterate functional decomposition over composed functions

State Partitioning

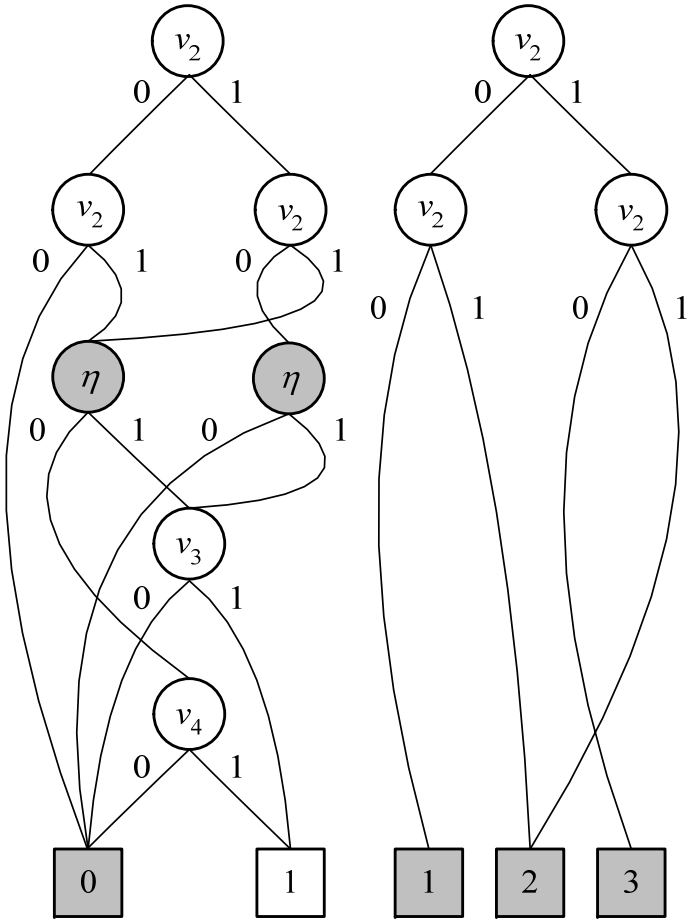
- BDD-based functional decomposition can be applied for state partitioning of a multiplexed FSM

$v_3 v_4$ \ $v_1 v_2$	0 0	0 1	1 0	1 1
0 0	0	0	0	0
0 1	0	1	0	1
1 0	0	0	0	0
1 1	0	1	0	1



State Partitioning

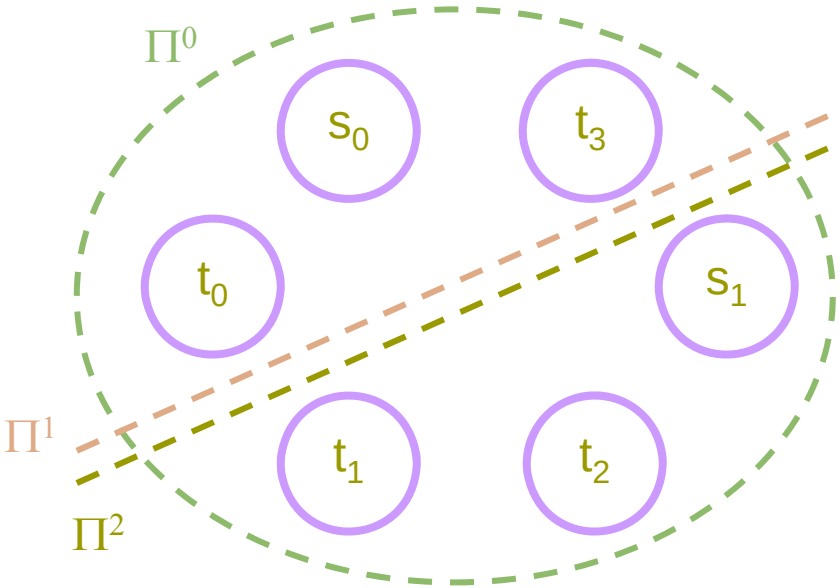
$\eta\, v_3\, v_4$ \ $v_1\, v_2$	0 0	0 1	1 0	1 1
0 0 0	0	0	0	0
0 0 1	0	1	0	1
0 1 0	0	0	0	0
0 1 1	0	1	0	1
1 0 0	0	0	0	0
1 0 1	0	0	0	0
1 1 0	0	1	1	1
1 1 1	0	1	1	1

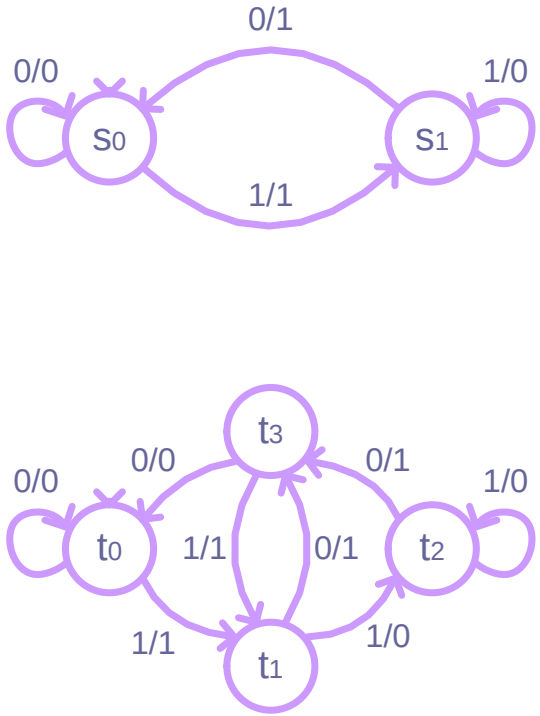


Multiple functions can be stacked using extra variables

State Partitioning

Example (cont'd)





Sequential EC

- Reachability analysis vs. state partitioning
  - Backward RA can be considered as state partitioning in the product state space

Exploiting Similarities for SEC

- Generic SEC
  - Works for checking designs with completely different circuit structures
  - Too hard due to state explosion
  - Designs under checking often possess similarities to some extent
- Desirable to reduce SEC to CEC as much as possible
  - Take advantage of structural similaritiesfor SEC

Register Correspondence

- Inductive register correspondence

  Base case: $I(\vec{s}) \Rightarrow R_{\underline{\underline{rc}}}(\vec{s})$, and

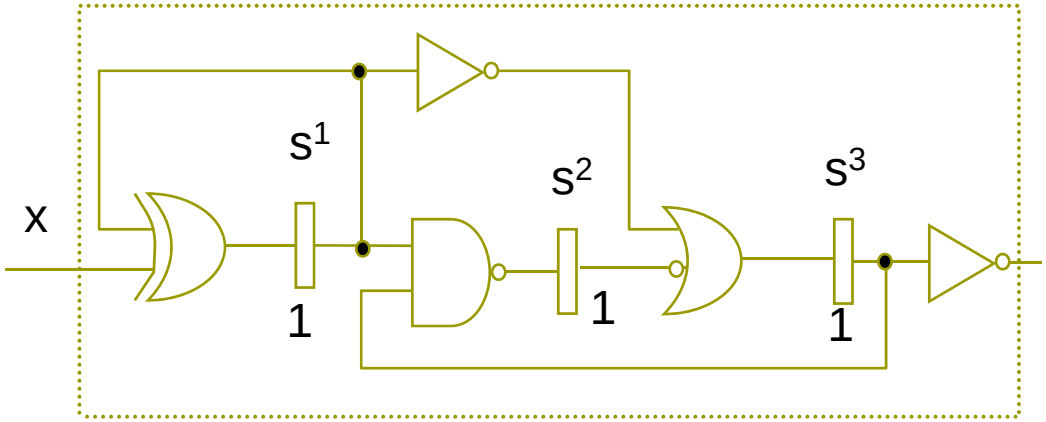
  Inductive case: $R_{\underline{\underline{rc}}}(\vec{s}) \Rightarrow R_{\underline{\underline{rc}}}(\vec{\delta}(\vec{x}, \vec{s}))$,

  where $R_{\underline{\underline{rc}}}(\vec{s}) = \bigwedge_{(s_i, s_j) \in \underline{\underline{rc}}} s_i \equiv s_j$

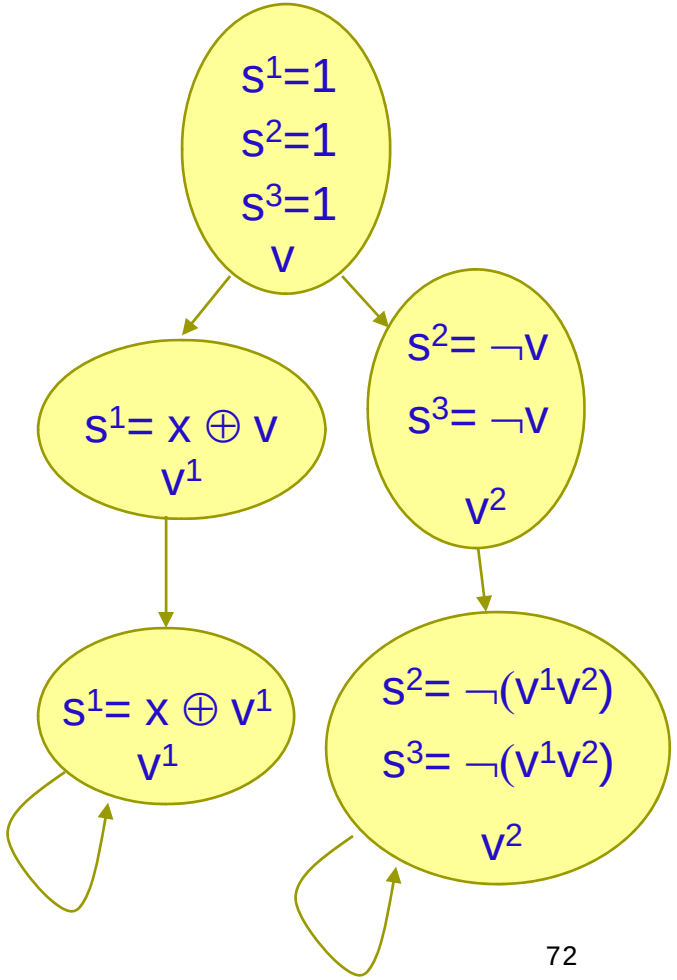
- Identify equivalence among registers not states
  - Computation scalable to large designs
- EC based on register correspondence is complete for circuits transformed by combinational synthesis

Register Correspondence

Example



Result: $\{s^1\}, \{s^2, s^3\}$



Signal Correspondence

- Inductive signal correspondence

  Base case: $I(\vec{s}) \Rightarrow R_{\underline{\underline{sc}}}(\vec{x}, \vec{s})$, and

  Inductive case: $R_{\underline{\underline{sc}}}(\vec{x}, \vec{s}) \Rightarrow R'_{\underline{\underline{sc}}}(\vec{x}, \vec{s})$,

  where $R_{\underline{\underline{sc}}}(\vec{x}, \vec{s}) = \bigwedge_{(f_i, f_j) \in \underline{\underline{sc}}} f_i(\vec{x}, \vec{s}) \equiv f_j(\vec{x}, \vec{s})$, and

  $R'_{\underline{\underline{sc}}}(\vec{x}, \vec{s}) = \bigwedge_{(f_i, f_j) \in \underline{\underline{sc}}} \forall x'. f_i(\vec{x}, \delta(\vec{x}, \vec{s})) \equiv f_j(\vec{x}, \delta(\vec{x}, \vec{s}))$

- Complete for retiming transformation

Safety Property Checking

Safety Property Checking

- Safety properties are the majority
  - For finite-state transition systems, liveness property checking can be converted to safety property checking
- Safety property checking can be formulated as reachability analysis

Model Checking

- Check if a state transition system M satisfies a temporal property φ
  - E.g. M |= $\varphi \equiv \mathrm{AG}(p \rightarrow \mathrm{AX}\ q)$
  - Equivalence checking is a special case
    - M : product machine
    - φ : every state reachable from the initial state has output label 0 under any transitions

      (a concise formula?)

Model Checking

- BDD-based model checking
  - So-called *symbolic model checking*
- SAT-based model checking
  - *Bounded model checking* (BMC)
    - Checking under a pre-specified length bound
  - *Unbounded model checking* (UMC)
    - Checking without length bound

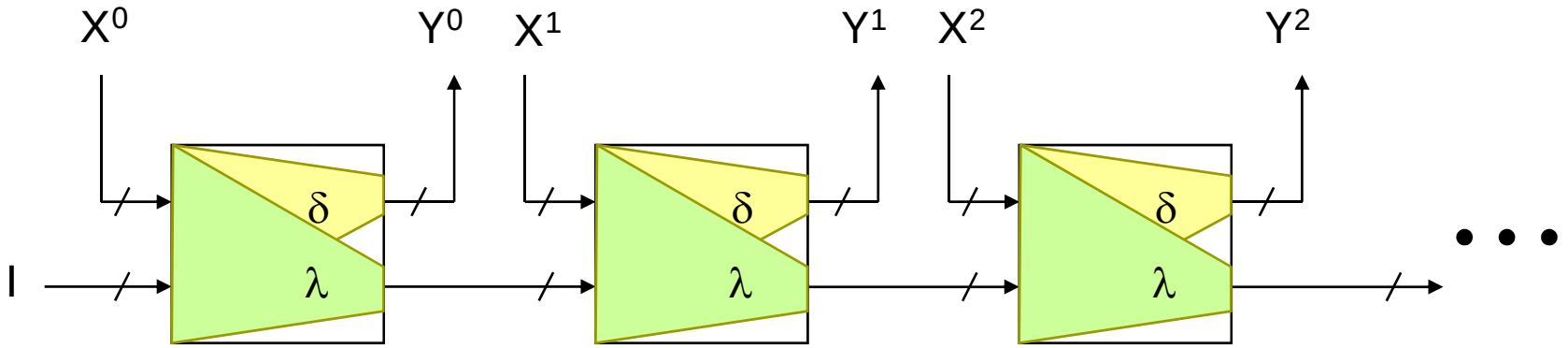
Symbolic Model Checking

- Safety property checking is formulated as reachability analysis
- Reachability analysis is done by BDD-based fixed-point computation

Bounded Model Checking

- Is any bad state reachable from the initial state in *k* steps?
  - Sound but not complete
  - *k* is bounded from above by the number of states (trivial bound; not useful in practice)
- Time-frame expansion
  - Similar to *automatic test pattern generation* (ATPG) technique in testing

Bounded Model Checking



E.g., in the context of SEC, check if the product machine can produce output 1 in *k* time-frames, for *k* = 1, 2, ...

Unbounded Model Checking

- Two approaches
  - By temporal induction
    - *k*-step induction
  - By Craig interpolation
    - Image approximation with interpolation

UMC with Temporal Induction

- Induction

  Base case: $I(\vec{s}) \Rightarrow P(\vec{s})$, and

  Inductive case: $P(\vec{s}) \wedge T(\vec{s}, \vec{s}') \Rightarrow P(\vec{s}')$

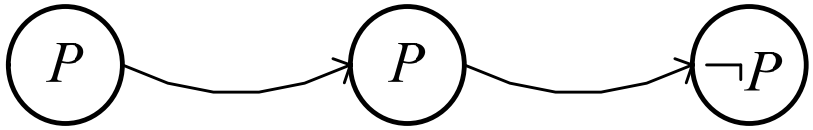
  - Incomplete whenever there is a P-state transition to a $\neg P$-state in the unreachable state space

UMC with Temporal Induction

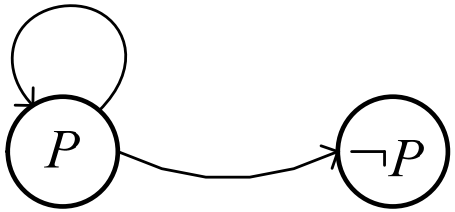
- k-step induction

Base case: $I(\vec{s}^0) \wedge T^k(\vec{s}^0,\ldots,\vec{s}^k) \Rightarrow P^k(\vec{s}^0,\ldots,\vec{s}^k)$, and

Inductive case: $P^k(\vec{s}^0,\ldots,\vec{s}^k) \wedge T^{k+1}(\vec{s}^0,\ldots,\vec{s}^{k+1}) \Rightarrow P(\vec{s}^{k+1})$



  - Still incomplete



UMC with Temporal Induction

- Simple-path criterion

$$\bigwedge_{1 \le j \le k} \vec{s}^{\,i} \neq \vec{s}^{\,j}$$

  - w/ simple-path criterion k-induction is complete
  - k is up-bounded by the length of the longest simple path

- Temporal induction can be implemented with incremental SAT solving

UMC with Craig Interpolation

- Over-approximated image computation using SAT
  - BMC + Craig interpolation allow us to compute image over-approximation relative to property.
    - Avoid computing exact image.
    - Take advantage of SAT solvers' strength of filtering out irrelevant facts.

UMC with Craig Interpolation

- Craig interpolation
  - Craig interpolation theorem [Cra57]:

    If $A \wedge B$ = **false**, there exists an *interpolant* A' for (A,B) such that

    1. $A \Rightarrow A'$
    2. $A' \wedge B$ = **false**
    3. A' refers only to common variables of A,B

    E.g. $A = p \wedge q$, $B = \neg q \wedge r$, $A' = q$

- Recent result
  - Given a resolution refutation of $A \wedge B$, A' can be derived in linear time.

UMC with Craig Interpolation

- Reachability analysis
  - Is there a state trajectory from I to F satisfying transition relation T ?
  - Reachability fixed point:

$$R_0 = I$$

$$R_{i+1} = R_i \vee Img(R_i, T)$$

$$R = \cup R_i$$

  - F is reachable from I iff $R \wedge F \neq \mathbf{false}$

UMC with Craig Interpolation

- Over-approximated reachability analysis

$$R'_0 = I$$
$$R'_{i+1} = R'_i \vee Img'(R'_i, T)$$
$$R' = \cup R'_i$$

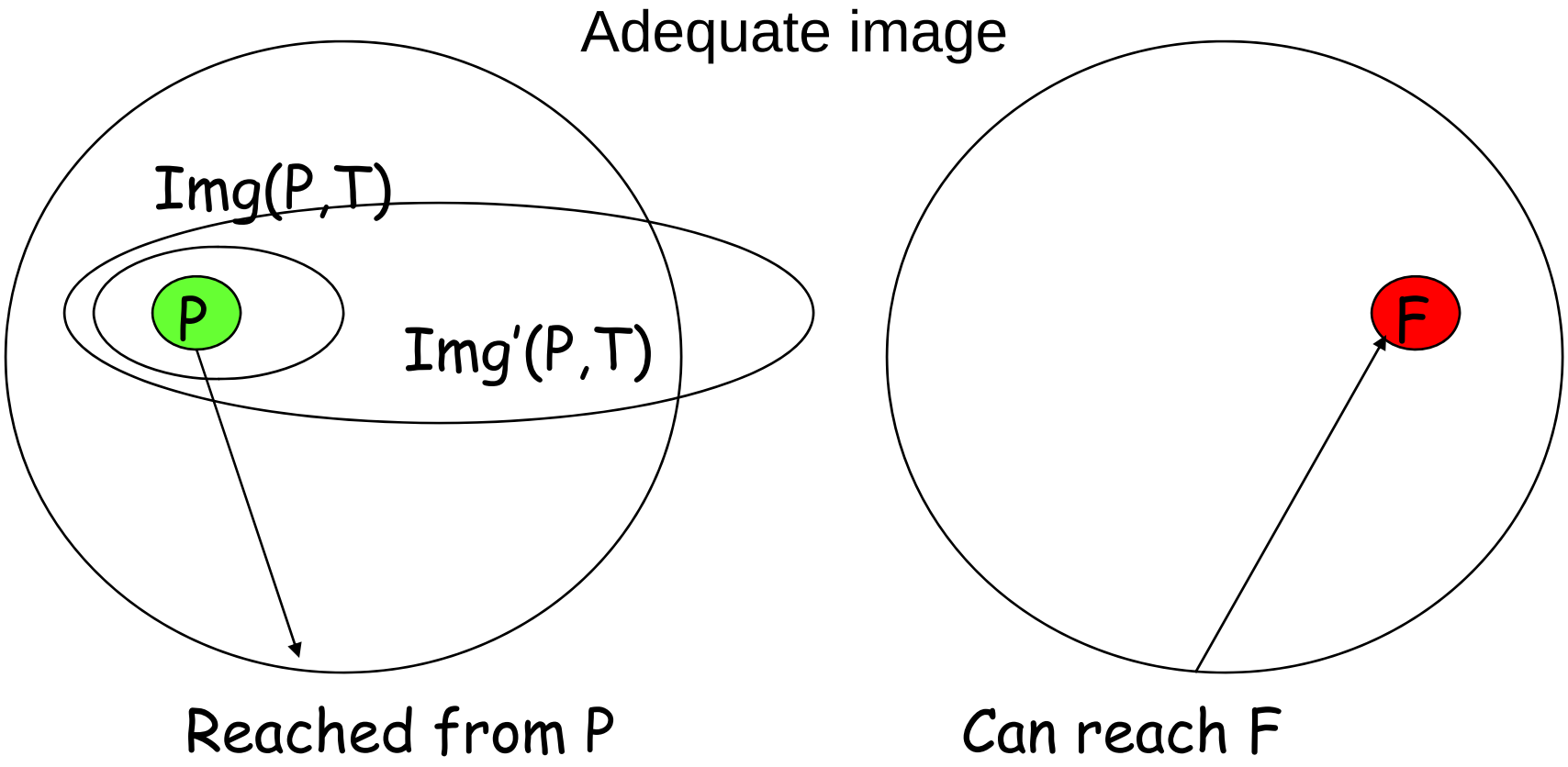
  - Img' is an over-approximate image operation s.t.

$$\forall P.\ \ Img(P, T) \Rightarrow Img'(P, T)$$

- Img' is *adequate* w.r.t. F, when
  if P cannot reach F, $Img'(P, T)$ cannot reach F

- If Img' is adequate, then
  F is reachable from I iff $R' \wedge F \neq \mathbf{false}$

UMC with Craig Interpolation



But how do you get an adequate Img'?

Source: McMillan's slides

UMC with Craig Interpolation

- k-adequacy (relaxed)
  - Img' is k-adequate w.r.t. F, when

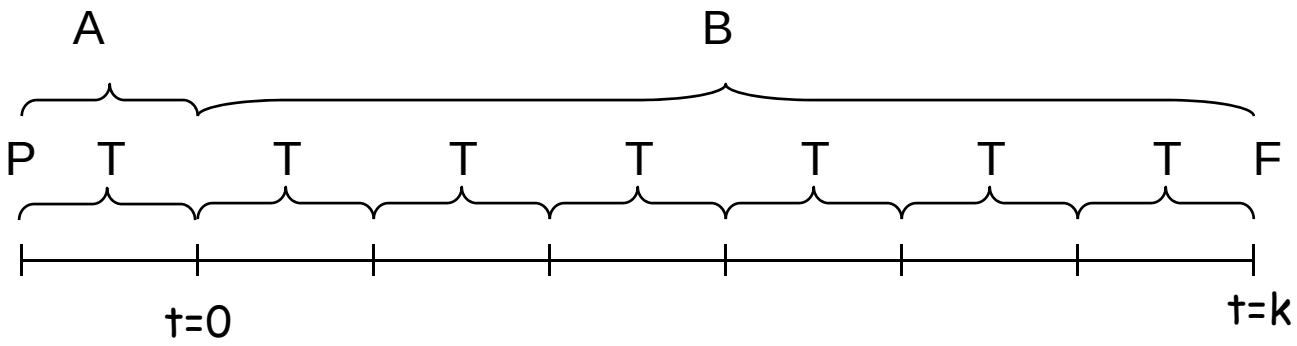
    if P cannot reach F, $Img'(P, T)$ cannot reach F *within* k *steps*
  - For k > (backward) diameter, k-adequate is equivalent to adequate.

UMC with Craig Interpolation

- Idea: use unfolding to enforce k-adequacy

$A = P_{-1} \wedge T_{-1}$

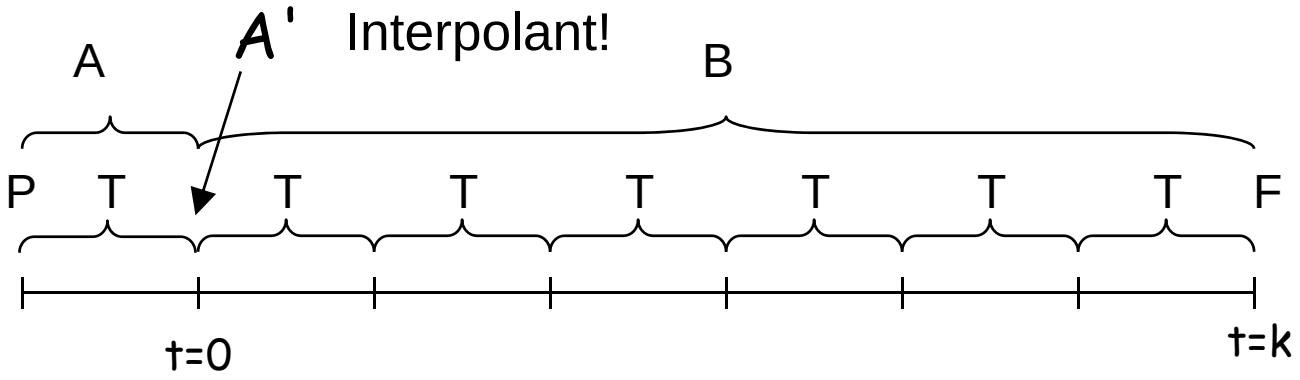
$B = T_0 \wedge T_1 \wedge \ldots \wedge T_{k-1} \wedge F_k$



Let $Img'(P)_0 = A'$,
where A' is an interpolant for (A,B)...

Img' is k-adequate!

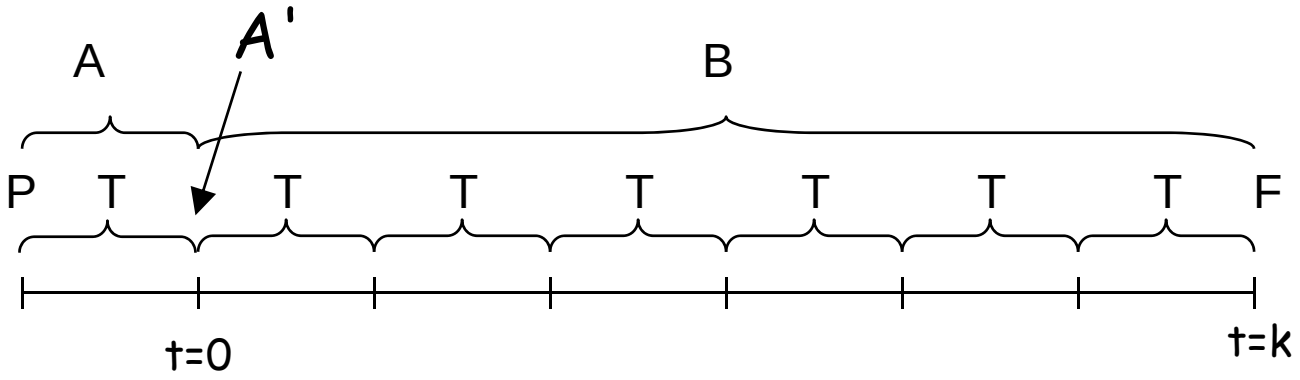
UMC with Craig Interpolation



- A $\Rightarrow$ A'
  - $Img(P, T) \Rightarrow Img'(P, T)$
- A' $\wedge$ B = **false**
  - $Img'(P, T)$ cannot reach F in k steps
- Hence Img' is k-adequate over-approximation. (Img' is undefined if A$\wedge$B is satisfiable.)

Source: McMillan's slides

UMC with Craig Interpolation



- Intuition
  - A' tells everything the SAT solver deduced about the image of P in proving it can't reach F in k steps.
  - Hence, A' is in some sense an abstraction of the image relative to the property.

UMC with Craig Interpolation

- Overall algorithm

  let k = 0

  repeat

    if I can reach F within k steps, answer reachable

    R = I

    while Img'(T, R) $\wedge$ F = **false**

      R' = Img'(T, R) $\vee$ R

      if R' = R answer unreachable

      R = R'

    increase k

UMC with Craig Interpolation

- Since k increases at every iteration, eventually k > d, the diameter, in which case Img' is adequate, and hence we terminate.

Notes:

- don't need to know when k > d in order to terminate (i.e. unbounded model checking)
- often termination occurs with k << d
- depth bound for temporal induction is the length of the longest simple path, which can be exponentially longer than diameter

Summary

- Computation basics
  - Characteristic functions and their manipulations
  - Data structures for Boolean reasoning
- Equivalence checking
  - Combinational and sequential EC
- Safety property checking
  - Bounded and unbounded model checking